

*Optimization, Backups, Replication, and more
High Performance MySQL*

第2版
涵蓋 Version 5.1



高性能 MySQL

*Baron Schwartz, Peter Zaitsev,
Vadim Tkachenko, Jeremy D. Zawodny,
Arjen Lentz & Derek J. Balling 著*
王小东 李军 康建勋 译

O'REILLY®



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

O'REILLY®

高性能 MySQL (第二版)

High Performance MySQL, 2nd Edition

Baron Schwartz Peter Zaitsev
Vadim Tkachenko Jeremy D. Zawodny 著
Arjen Lentz Derek J. Balling

王小东 李军 康建勋 译

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

www.TopSage.com

内容简介

本书荣获 2009 年 Jolt 图书大奖,是不可多得的分享 MySQL 实用经验的图书。它不但可以帮助 MySQL 初学者提高使用技巧,更为有经验的 MySQL DBA 指出了开发高性能 MySQL 应用的途径。全书包含 14 章和 4 个附录,内容覆盖 MySQL 系统架构、设计应用技巧、SQL 语句优化、服务器性能调优、系统配置管理和安全设置、监控分析,以及复制、扩展和备份/还原等主题,每一章的内容自成体系,适合各领域技术人员作选择性的阅读。

978-0-596-10171-8 High Performance MySQL, Second Edition © 2008 by O'Reilly Media, Inc. Simplified Chinese edition, jointly published by O'Reilly Media, Inc. and Publishing House of Electronics Industry, 2010. Authorized translation of the English edition, 2008 O'Reilly Media, Inc., the owner of all rights to publish and sell the same. All rights reserved including the rights of reproduction in whole or in part in any form.

本书中文简体版专有出版权由 O'Reilly Media, Inc. 授予电子工业出版社,未经许可,不得以任何方式复制或抄袭本书的任何部分。

版权贸易合同登记号 图字:01-2008-4026

图书在版编目(CIP)数据

高性能 MySQL: 第 2 版 / 施瓦茨 (Schwartz, B.) 等著; 王小东, 李军, 康建勋译. —北京: 电子工业出版社, 2010.1
书名原文: High Performance MySQL, 2nd Edition
ISBN 978-7-121-10245-5

I. 高… II. ①施…②王…③李…④康… III. 关系数据库-数据库管理系统, MySQL IV. TP311.138

中国版本图书馆 CIP 数据核字 (2010) 第 010252 号

策划编辑: 徐定翔

责任编辑: 周 筠

项目管理: 梁 晶

封面设计: Karen Montgomery, 张 健

印 刷: 北京天宇星印刷厂

装 订: 三河市皇庄路通装订厂

出版发行: 电子工业出版社

北京市海淀区万寿路173信箱 邮编: 100036

开 本: 860×1092 1/16 印张: 35 字数: 950千字

印 次: 2010年1月第1次印刷

定 价: 99.00元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系,

联系及邮购电话: (010) 88254888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线: (010) 88258888。

O'Reilly Media, Inc.介绍

为了满足读者对网络和软件技术知识的迫切需求，世界著名计算机图书出版机构 O'Reilly Media, Inc. 授权电子工业出版社，翻译出版一批该公司久负盛名的英文经典技术专著。

O'Reilly Media, Inc. 是世界上在Unix、X、Internet和其他开放系统图书领域具有领导地位的出版公司，同时也是在线出版的先锋。

从最畅销的《The Whole Internet User's Guide & Catalog》（被纽约公共图书馆评为20世纪最重要的50本书之一）到GNN（最早的Internet 门户和商业网站），再到 WebSite（第一个桌面 PC 的Web服务器软件），O'Reilly Media, Inc.一直处于Internet发展的最前沿。

许多书店的反馈表明，O'Reilly Media, Inc.是最稳定的计算机图书出版商——每一本书都一版再版。与大多数计算机图书出版商相比，O'Reilly Media, Inc. 具有深厚的计算机专业背景，这使得O'Reilly Media, Inc. 形成了一个非常不同于其他出版商的出版方针。O'Reilly Media, Inc. 所有的编辑人员以前都是程序员，或者是顶尖级的技术专家。O'Reilly Media, Inc.还有许多固定的作者群体——他们本身是相关领域的技术专家、咨询专家，而现在编写著作，O'Reilly Media, Inc.依靠他们及时地推出图书。因为 O'Reilly Media, Inc.紧密地与计算机业界联系着，所以 O'Reilly Media, Inc. 知道市场上真正需要什么图书。

在进入 MySQL 世界之前，先照例介绍一下 MySQL 的历史（况且本书里也没提到这些）。

真正以 MySQL 为名的数据库是从 1994 年开始开发的，并于 1995 年第一次呈现在小范围的用户面前，它的开发者刚好不是美国人，而是两个瑞典人 Michael Widenius 和 David Axmark。那时的 MySQL 还非常简陋，除了在一个表上做一些 Insert、Update、Delete 和 Select 操作，恐怕没有更多的功能给用户使用。这种情况直到 2001 年左右发布 3.23 版的时候，才有了显著的进步——它支持大多数的基本 SQL 操作了，而且还集成了我们现在熟识的 MyISAM 和 InnoDB 存储引擎。然后又是几年不断完善的过程，到了 2004 年 10 月，这个夯实基础的过程到达了顶峰——4.1 这个经典版本发布了。次年 10 月，又一里程碑式的 MySQL 版本发布了，在新出的 MySQL 5.0 里加入了游标、存储过程、触发器、视图和事务的支持，准备进入中高端应用领域。在 5.0 之后的版本里，MySQL 明确地表现出迈向高性能数据库的发展步伐。

到今天，MySQL 已经上升到了 600 多万的装机量，著名的 WordPress、phpBB 都以 MySQL 为后台数据库，很多大型的 WWW 应用例如 Wikipedia、Google 和 Facebook，也都采用了 MySQL 作为它们的数据存储系统。

反观国内，鉴于心照不宣的原因，MySQL 的普及程度还不如 SQL Server。就我这些年来的所见所闻而言，一直作为 MySQL 黄金搭档的 PHP 都常常使用别的数据库，更别提其他开发语言了。好在那些上规模企业，尤其是外资企业里，多数明智的 IT 负责人在项目前期都会提议使用 MySQL，原因之一是它是免费的，一般不会产生授权费用问题，原因之二是它足够用了，不是吗？你想要的增、删、改、联接（Joint）、嵌套查询它都有；你想要的视图、存储过程、触发器、事务它也有；如果你要集群，它也能提供。

但是，使用 MySQL 是一回事，用好 MySQL 又是另外一回事。市面上更多的是关于 MySQL 开发的书籍，这些书籍的很多篇幅都花费在 SQL 语句的学习上。若要获得关于 MySQL 性能提高方面的资料，我们只能在网上的各个论坛或博客上披沙拣金了，而本书则系统性地从各个方面讲述一个高性能 MySQL 应用应该怎么来做。作者们都是这方面的行家里手，所以内容也是全面、充实，无论是架构师设计师、程序开发人员，还是系统管理员都能找到感兴趣的方面。在阅读正文前，最好能够先读一下作者精心编写的前言部分，通过它把握整本书内容的构成方式和相互关联，之后，带有目的性地阅读本书会更富有成效。

本书由李军、王小东、康建勋三人合作翻译完成，其中，康建勋翻译第 1 章和第 2 章前 31 页；王小东翻译第 2 章的后 17 页，以及第 3 章至第 8 章；李军翻译了序言、前言、第 9 章至第 14 章、所有附录，以及作者介绍、封面、封底等内容，并撰写了内容简介。翻译的过程也是译者与编辑、审阅人员之间交互的过程，在这个过程中，编辑徐定翔老师，审阅人金照林老师、柳安意老师给予了我们很大的帮助。如果说译者是生产毛坯的工匠，那么他们就是把毛坯打磨成精品呈现给读者的人，在此十分感谢他们！

同样地，我们也要感谢家人和朋友。我们把那些本来应该陪伴家人出游，或者参加朋友聚会的时间，都“自私”地用在翻译本书上了。他们都比较宽容，一句“到时要请客哦”就原谅了我们，谢谢他们的支持！

最后，得向读者们说声抱歉，由于术业专攻不同、识见浅深有别之故，译文中难免会有诘屈聱牙、词不达意甚至疏误之处，还请读者不吝指正（译注 1）。

译者
2009 年 12 月

译注 1：由于本书篇幅较大，为了节约成本和便于读者阅读，我们将原书版式作了压缩，原书页码用“”表示，供读者对照。本书的索引（包括正文中的交叉索引）所列页码为原英文版页码。

| | |
|----------------------------|-----|
| 推荐序 | I |
| 前言 | III |
| 第 1 章 MySQL 架构 | 1 |
| 1.1 MySQL 的逻辑架构 | 1 |
| 1.2 并发控制 | 3 |
| 1.3 事务 | 4 |
| 1.4 多版本并发控制 | 10 |
| 1.5 MySQL 的存储引擎 | 11 |
| 第 2 章 寻找瓶颈：基准测试与性能分析 | 25 |
| 2.1 为什么要进行基准测试 | 25 |
| 2.2 基准测试策略 | 26 |
| 2.3 基准测试工具 | 32 |
| 2.4 基准测试样例 | 34 |
| 2.5 性能分析 (Profiling) | 43 |
| 2.6 分析操作系统 | 60 |
| 第 3 章 架构优化和索引 | 63 |
| 3.1 选择优化的数据类型 | 63 |
| 3.2 索引基础知识 | 74 |
| 3.3 高性能索引策略 | 83 |
| 3.4 索引实例研究 | 102 |
| 3.5 索引和表维护 | 105 |
| 3.6 正则化和非正则化 | 108 |
| 3.7 加速 ALTER TABLE | 113 |
| 3.8 对存储引擎的说明 | 115 |
| 第 4 章 查询性能优化 | 118 |
| 4.1 基本原则：优化数据访问 | 118 |
| 4.2 重构查询的方式 | 122 |
| 4.3 查询执行基础知识 | 124 |

| | | |
|-------|---------------------------------|-----|
| 4.4 | MySQL 查询优化器的限制 | 139 |
| 4.5 | 优化特定类型的查询 | 146 |
| 4.6 | 查询优化提示 | 151 |
| 4.7 | 用户定义变量 | 154 |
| 第 5 章 | MySQL 高级特性 | 159 |
| 5.1 | MySQL 查询缓存 | 159 |
| 5.2 | 在 MySQL 中存储代码 | 168 |
| 5.3 | 游标 | 173 |
| 5.4 | 准备语句 | 174 |
| 5.5 | 用户自定义函数 | 177 |
| 5.6 | 视图 | 179 |
| 5.7 | 字符集和排序规则 | 182 |
| 5.8 | 全文搜索 | 188 |
| 5.9 | 外键约束 | 194 |
| 5.10 | 合并表和分区 | 194 |
| 5.11 | 分布式 (XA) 事务 | 201 |
| 第 6 章 | 优化服务器设置 | 203 |
| 6.1 | 配置基础知识 | 203 |
| 6.2 | 通用调优原则 | 207 |
| 6.3 | MySQL I/O 调优 | 214 |
| 6.4 | MySQL 并发调优 | 224 |
| 6.5 | 基于工作负载调优 | 226 |
| 6.6 | 每连接 (Per-Connection) 设置调优 | 231 |
| 第 7 章 | 操作系统和硬件优化 | 232 |
| 7.1 | 什么限制了 MySQL 的性能 | 232 |
| 7.2 | 如何为 MySQL 选择 CPU | 233 |
| 7.3 | 平衡内存和磁盘资源 | 235 |
| 7.4 | 为从服务器选择硬件 | 240 |
| 7.5 | RAID 性能优化 | 240 |
| 7.6 | 存储区域网络和网络附加存储 | 246 |
| 7.7 | 使用多个磁盘卷 | 247 |
| 7.8 | 网络配置 | 248 |
| 7.9 | 选择操作系统 | 250 |
| 7.10 | 选择文件系统 | 250 |
| 7.11 | 线程处理 | 252 |
| 7.12 | 交换 | 252 |
| 7.13 | 操作系统状态 | 254 |

| | | |
|--------|----------------------------|-----|
| 第 8 章 | 复制 | 259 |
| 8.1 | 复制概述 | 259 |
| 8.2 | 创建复制 | 262 |
| 8.3 | 揭示复制的真相 | 268 |
| 8.4 | 复制拓扑 | 273 |
| 8.5 | 复制和容量规划 | 284 |
| 8.6 | 复制管理和维护 | 285 |
| 8.7 | 复制问题和解决方案 | 292 |
| 8.8 | 复制有多快 | 305 |
| 8.9 | MySQL 复制的未来 | 307 |
| 第 9 章 | 伸缩性与高可用性 | 308 |
| 9.1 | 术语 | 308 |
| 9.2 | MySQL 的伸缩性 | 310 |
| 9.3 | 负载均衡 | 328 |
| 9.4 | 高可用性 | 336 |
| 第 10 章 | 应用层面的优化 | 344 |
| 10.1 | 应用程序性能概述 | 344 |
| 10.2 | Web 服务器的议题 | 346 |
| 10.3 | 缓存 | 349 |
| 10.4 | 扩展 MySQL | 354 |
| 10.5 | 可替代的 MySQL | 354 |
| 第 11 章 | 备份与还原 | 356 |
| 11.1 | 概况 | 356 |
| 11.2 | 要权衡的事项 | 360 |
| 11.3 | 管理和备份二进制日志 | 367 |
| 11.4 | 数据备份 | 369 |
| 11.5 | 从备份中还原 | 377 |
| 11.6 | 备份和还原的速度 | 386 |
| 11.7 | 备份工具 | 387 |
| 11.8 | 脚本化备份 | 392 |
| 第 12 章 | 安全 | 395 |
| 12.1 | 术语 | 395 |
| 12.2 | 账号的基本知识 | 396 |
| 12.3 | 操作系统安全 | 411 |
| 12.4 | 网络安全 | 412 |
| 12.5 | 数据加密 | 418 |
| 12.6 | 在 Chroot 环境里使用 MySQL | 421 |

第 13 章 MySQL 服务器的状态.....423

13.1 系统变量 423

13.2 SHOW STATUS 423

13.3 SHOW INNODB STATUS..... 429

13.4 SHOW PROCESSLIST 440

13.5 SHOW MUTEX STATUS 441

13.6 复制的状态 442

13.7 INFORMATION_SCHEMA 442

第 14 章 用于高性能 MySQL 的工具.....444

14.1 带界面的工具 444

14.2 监控工具 446

14.3 分析工具 453

14.4 MySQL 的辅助工具 455

14.5 更多的信息来源 458

附录 A 大文件传输459

附录 B 使用 EXPLAIN.....463

附录 C 在 MySQL 里使用 Sphinx.....476

附录 D 锁的调试.....497

索引505

推荐序

TopSage.com

我认识 Peter、Vadim 和 Arjen 已经有很长一段时间，见证了他们长久以来在自己项目上使用 MySQL 和为各类高标准客户调优 MySQL 服务器的历史。另一方面，Baron 为增强 MySQL 的功能编写了许多客户端软件。

作者们的专业背景清晰地反映在了彻底重写《High Performance MySQL: Optimizations, Replication, Backups, and More》第二版的工作里。这本书不只告诉你如何优化工作，从而能比以前更好地使用 MySQL，作者们还做了大量额外的工作，亲自编制执行基准测试，并将结果发布出来以佐证他们的观点。这些信息让读者可以借此获悉许多很有价值的 MySQL 内部工作机制——这在其他书中是难以得到的；同样，这些信息也能帮助读者避开那些在将来会引发糟糕性能的错误。

我不但要向刚刚接触 MySQL 服务器，正准备编写第一个 MySQL 应用的初学者推荐这本书，还要向富有经验的用户推荐这本书，他们已经对基于 MySQL 的应用作过一些调优的工作，现在正需要在这个方向上再前进“一小步”。

——Michael Widenius

2008 年 3 月

对于这本书，我们在头脑里有好几个目标。其中的大多数源于我们一直想要有一本在书架上寻找却总是找不到的神话般完美的 MySQL 书，其他几个目标来自于想把我们的经验分享给那些把 MySQL 用在他们环境中的用户。

我们不想让这本书只是一本 SQL 入门书，不想让这本书的书名随意使用一些时限词语来开始或结尾（例如“……只需 30 天”，“7 天内提高……”），也不想说服读者什么。最主要的，我们希望这本书能帮助你把技能提高一个层次，用 MySQL 构建出快速、可用的系统——它能解答类似这样的问题：“我怎样才能搭建起一个 MySQL 服务器集群，它能处理数以百万计的请求，哪怕有几台服务器宕机时，它仍然能正常提供服务？”

我们编写本书的着眼点不仅在于迎合 MySQL 应用开发人员的需求，还在于满足 MySQL 管理员的严格要求，管理员需要不管开发人员和用户怎么折腾，服务器都能挂在线上正常运行。如前所述，我们假定你已经具备了一些 MySQL 的相关经验，比较理想的就是你已经读过一本 MySQL 方面的入门书。我们同样也假定你具备一些常用的系统管理、网络和 Unix 风格操作系统等方面的经验。

经过修订、扩充后的第二版对于第一版里的所有主题都作了更深入的讲解，并增加了一些新的主题。这也部分地反映了自本书首次出版之后，MySQL 世界发生的一些变化：MySQL 现在已经成为软件中更大更复杂的一部分。如同其重要性一样，它的普及度也提高了：MySQL 社区变得更加庞大，更多的大企业把 MySQL 应用到他们的关键业务系统中。自本书第一版发布以后，MySQL 已被广泛认同可作为企业级应用（注 1）。人们也越来越地把 MySQL 用在互联网应用上，这些应用若发生故障和其他问题都无法被掩饰过去，也不能被容忍。

作为我们努力的结果，第二版的内容着重点跟第一版略有不同。我们会像强调性能一样，强调可用性和准确性，这部分由于我们自己也把 MySQL 用在那些运作着巨大金额的业务系统里。我们对 Web 应用也有着切身体验，MySQL 在这方面正变得越来越普及。第二版里会谈论到在 MySQL 周边扩展的世界，而这个世界在第一版编写时还不存在。

本书是如何组织的

How this book is organized

我们把许多复杂的主题放在一本书里，所以，在这里我们要解释一下它们的编排次序，使读者能更易于学习它们。

注 1：我们觉得这段话更像是市场营销的说辞，但是，它大概传达了这样一个意思：MySQL 对于许多人而言显得很重要了。

内容广泛的概述

A Broad Overview



第1章，MySQL 架构，用于讲述基础知识——这些知识在你做更深入挖掘之前必须加以熟悉。你需要在高效利用 MySQL 前理解整个框架是如何被组织起来的。这一章解释了 MySQL 的架构和它存储引擎的关键方面。如果你还不熟悉关系数据库的一些基本概念及事务，它就能帮你更快地进入角色。如果本书就是你的 MySQL 入门书，这一章也非常有用，不过，你最好还是事先熟悉了另外一种数据库，例如 Oracle。

构建一个坚实的基础

Building a Solid Foundation

接下来的4章涉及了你在使用 MySQL 时会几次三番来查阅的资料。

第2章，寻找瓶颈：基准测试与性能分析，讨论了基准测试和获取系统概况的基础。它们决定了你的系统能处理哪一类的工作负荷、执行某些任务时它能运行得多快等。你会希望在做重要更改的前后都能对你的应用做一次基准测试，这样就可以判断出这些更改产生了多大的效果。有些看似正面的更改在真实世界的负载压力下可能会变成负面的影响，除非你能精确地对其进行衡量，否则你是永远都不会知道到底是什么导致糟糕的系统性能。

第3章，架构优化和索引，我们会介绍各数据类型的细微差别、表的设计和索引的创建。一个设计良好的数据库能有助于 MySQL 获得更佳的性能表现。在接下来的那些章节里，我们讲到的很多内容的关键点都在于你的应用是怎么使用 MySQL 索引的。深刻认识索引及如何巧妙地运用它们是高效使用 MySQL 的核心所在，所以，你可能会经常地回过头来重新阅读这一章。

第4章，查询性能优化，解释了 MySQL 是怎样执行查询的，以及怎么才能利用查询优化器的能力。深入领会查询优化器的工作方法能帮你在编写查询时创造奇迹，也能帮你更好地理解索引（索引和查询优化的次序就像“先有鸡还是先有蛋”的问题，读完第4章后再回去读第3章可能会让你受益匪浅。）该一章里还特别展示了那些常见的查询示例，用来说明 MySQL 所擅长的是哪方面的工作，怎么把查询转换成能够利用查询优化器强大能力的形式。

为了更好地做到这一点，我们已经讲述过对任何数据库都适用的一些基本概念：表、索引、数据和查询。第5章，MySQL 高级特性，将在上述基础之上再进一步，向你展示 MySQL 内部那些更高层次的框架是如何运作的。我们会介绍查询缓存、存储过程、触发器、字符集等内容。MySQL 实现这些功能特性的方法跟其他数据库有点不一样，因此，对这些特性的深入理解能够帮你创造一个性能优化的新机会，这在以前你是不会想到的。

调优你的应用

Tuning Your Application

接下来的两章讨论如何修改你的基于 MySQL 的应用，使它能在性能上得到提升。

第6章，优化服务器设置，我们讨论的是如何调优 MySQL，使它能在最大程度上让硬件特性为你的特定应用服务。第7章，操作系统和硬件优化，我们解释了如何充分利用你的操作系统和硬件配置，同时为大规模应用提供了某些能提高性能的硬件配置建议。

配置更改之后的向上扩展

Scaling Up MySQL After Schema Changes

一台服务器往往是不够用的。第 8 章，复制，介绍如何把数据自动地复制到多台服务器上。第 9 章，“伸缩性与高可用性”，讲述如何将伸缩性、负载均衡和高可用性综合起来运用，为应用伸展到你所需要的程度提供基础性工作。

当应用运行在一个大规模的 MySQL 后端之上时，它本身就蕴含了意义非凡的优化机会。设计一个大型应用有更好的途径，也有更坏的途径，但这不是本书的着重点，我们不希望你把所有的时间都专注于 MySQL 之上。第 10 章，“应用层面的优化”，帮助你发现那些悬挂在靠近地面枝头上的柿子，特别是对于 Web 应用。

增强应用的可靠性

Looking for Application Reliability

哪怕是世界上设计得最好、伸缩性最强的架构，如果它不能在掉电、恶意攻击、程序 Bug、程序员的过失，以及其他灾难中幸存下来，那它也算不上是好的架构。

第 11 章，“备份和还原”，我们会讨论到不同的 MySQL 数据库备份和还原策略。这些策略都有助于在系统遭受到不可避免的硬件错误时最小化故障停机时间，遭遇到各种灾难时确保你的数据安全。

第 12 章，“安全”，能让你对运行 MySQL 服务器涉及的安全因素有深入的认识。最重要的是，我们给你提供了很多建议，防止来自外部的攻击威胁你苦心优化、配置过的服务器。我们还会指出几个很少见的暴露出数据库安全问题的地方，并展示不同的实施方法的好处及对性能的影响。通常情况下，就性能方面而言，保持安全策略简单化是值得的。

其他有用的主题

Some Other Useful Topics

最后的几个章节和附录里，我们深入研究了几个既不“适合”放入前面任何一个章节中，又被多个章节反复引用的内容，它们值得特别关注。

第 13 章，“MySQL 服务器的状态”，展示的是如何检查 MySQL 服务器运行情况。知道如何获取服务器的状态信息很重要，知道那些信息包含的意思更加重要。我们针对 SHOW INNODB STATUS 作了特别具体的讲解，它能提供关于 InnoDB 事务存储引擎的更深层次的操作信息。

第 14 章，“用于高性能的 MySQL 工具”，介绍了一些能帮你更有效管理 MySQL 的工具。这些工具包括监控和分析工具，以及能帮你编写查询语句的工具等。其中提到的 Maatkit 是由 Baron 创建的，它能够增强 MySQL 的功能性，使你的数据库管理员的生活更加轻松。在该章里也演示了一个名叫 innotop 的程序，这个程序是 Baron 写的，其目的是提供一个易于使用的查看 MySQL 正在做什么的用户接口，它的功能与 Unix 的 top 命令类似。在调优 MySQL 各阶段里，你若监控 MySQL 和它的存储引擎里发生的情况，它就是一个很有价值的工具。

附录 A，“大文件传输”，展示如何高效地把很大的文件从一个地方复制到另一个地方——这在大数据量管理时肯定会用到。附录 B，“使用 EXPLAIN”，展示如何真正理解和使用那个重要的 EXPLAIN 命令。附录 C，“在 MySQL 里使用 Sphinx”是对 Sphinx 的一个介绍，这个高性能全文索引系统是对 MySQL 自有功能的一个补充。

最后的附录 D,“锁的调试”,向你展示的是当几个查询在请求锁时相互妨碍时,该如何去破译其中的缘由。

软件的版本和有效性

Software Versions and Availability

MySQL 就像个移动的目标。在 Jeremy 写出第一版提纲之后的几年里,有大量的不同版本 MySQL 发布出来。在本书第一版发行的时候,MySQL 4.1 和 5.0 还都是 alpha 版,但如今它们已经作为正式产品很多年了,并成为今天许多大型在线应用的后台支撑。当我们完成本书第二版时,MySQL 5.1 和 6.0 也处于这样的边缘。(MySQL 5.1 是 candidate 版,而 6.0 是 alpha 版。)

在本书里,我们没有依赖于某个特定的 MySQL 版本。相反,我们讲述的是基于真实世界里各版本 MySQL 的更广阔的知识。本书涉及的核心版本是 MySQL 5.0,因为我们把它看作是“当前”版本。书中的大多数示例都假定你运行的是 MySQL 5.0 的某个相对比较稳定的版本,例如 MySQL 5.0.40 或更新的。我们会特意标注出哪些框架或功能在那些老版本里不存在,或者会出现在即将到来的 5.1 版本系列里。然而,明确的功能特性与版本的对应关系只有在 MySQL 的文档里才能找到,所以,我们希望你阅读本书的时候能够经常访问带有注解的在线文档 (<http://dev.mysql.com/doc/>)。

MySQL 另一个伟大之处在于它能运行在当今所有流行的平台上:Mac OS X、Windows、GNU/Linux、Solaris、FreeBSD,只要你能想到的都行!但是,我们偏向于 GNU/Linux (注 2) 和其他 Unix 风格的操作系统。Windows 用户可能会有所不同,例如,文件路径就会完全不一样。书中也会用到一些标准的 Unix 命令行功能,我们假定你知道它们在 Windows 上的对应命令。(注 3)

Perl 也是 MySQL 在 Windows 上运行时的麻烦之一。MySQL 自带的几个很有用的辅助功能都是用 Perl 写的,本书某一些章节里展示的 Perl 脚本就是构建更复杂工具的基础,像 Maatkit 也是用 Perl 写的。可是,Perl 并没有包含在 Windows 里。为了能使用这些脚本,你需要访问 ActiveState 下载一个 Windows 版的 Perl,然后安装一个必需的插件模块 (DBI 和 DBD::mysql) 好让 MySQL 能够访问到它。

本书使用的书写约定

Conventions Used in This Book

本书使用了以下这些书写约定:

等宽字体 (Constant width)

用于表示代码、配置选项、数据库和表名、变量和它们的值、函数、模块、文件内容,以及命令的输出结果。

等宽粗体 (Constant width Bold)

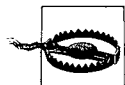
用于表示命令或要用户自己输入的内容,也用于强调命令的输出结果里的某些内容。

注 2: 为了避免混淆,当我们写到关于内核的内容时,就以 Linux 称呼;当我们讲到支持应用的整个操作系统架构时,就以 GNU/Linux 称呼。

注 3: 你可以在 <http://unxutils.sourceforge.net> 或 <http://gnuwin32.sourceforge.net> 下载到与 Windows 兼容的 Unix 辅助工具。



这个图标表示提示、建议或一般性注解。



这个图标表示的是提醒或警告。

使用本书示例代码

本书的目的是帮你把事情做好。一般来说，你无需特地联系我们就可以在你的程序和文档里任意使用本书的代码，除非你要把其中的关键代码以你的名义重新发布。举例来说，你的程序中使用到了书中的几段代码，不需要获得许可；出售或发布 O'Reilly 的随书光盘，需要获得许可；引用本书内容和示例代码去解答一个问题，不需要获得许可；把本书中大量代码合并到你的产品文档里时，需要获得许可。

本书的示例代码在 <http://www.highperformmysql.com> 上可以获取到，并经常会有更新。但是，我们不能保证会为所有次要版本的 MySQL 更新和测试这些代码。

我们会感谢，但是不要求写上代码所属权的声明。这个所有权声明一般包括书名、作者、出版商和 ISBN，例如“High Performance MySQL: Optimization, Backups, Replication, and More, Second Edition, by Baron Schwartz et al. Copyright 2008 O'Reilly Media, Inc., 9780596101718.”

如果你觉得你对示例代码的使用超过了正当使用范围或如上所述的授权使用的范围了，请跟我们联系：permissions@oreilly.com。

如何联系我们

我们已尽力核验本书所提供的信息，尽管如此，仍不能保证本书完全没有瑕疵，而网络世界的变化之快，也使得本书永不过时的保证成为不可能。如果读者发现本书内容上的错误，不管是赘字、错字、语意不清，甚至是技术错误，我们都竭诚虚心接受读者指教。如果您有任何问题，请按照以下的联系方式与我们联系。

奥莱理软件（北京）有限公司
北京市 西城区 西直门 南大街2号 成铭大厦C座807室
邮政编码：100080
网页：<http://www.oreilly.com.cn>
E-mail：info@mail.oreilly.com.cn

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international/local)
707-829-0104 (fax)

与本书有关的在线信息如下所示。

<http://www.oreilly.com/catalog/9780596101718> (原书)

<http://www.oreilly.com.cn/book.php?bn=978-7-121-10245-5> (中文版)

北京博文视点资讯有限公司 (武汉分部)

湖北省 武汉市 洪山区 吴家湾 邮科院路特1号 湖北信息产业科技大厦1402室

邮政编码: 430074

电话: (027)87690813 传真: (027)87690813转817

读者服务网页: <http://bv.csdn.net>

E-mail:

reader@broadview.com.cn (读者信箱)

bvtougao@gmail.com (投稿信箱)

本书第二版的致谢

Acknowledgments for the Second Edition

Sphinx 的开发者 Andrew Aksyonoff 编写了附录 C, “在 MySQL 里使用 Sphinx”。我们非常感谢他首次对此进行深入讨论。

在本书编写的时候, 我们也得到了来自于许多人的无私帮助, 在这里我们不可能把他们都一一列举出来——我们真地非常感谢 MySQL 社区和 MySQL AB 公司的每一个人。下面是对本书作出了直接贡献的人, 如果我们遗漏了任何一个人, 还请原谅, 他们是: Tobias Asplund、Igor Babaev、Pascal Borghino、Roland Bouman、Ronald Bradford、Mark Callaghan、Jeremy Cole、Britt Crawford 和他的 HiveDB 项目、Vasil Dimov、Harrison Fisk、Florian Haas、Dmitri Joukovski、Zmanda (感谢他为解释 LVM 快照配上的图表)、Alan Kasindorf、Sheeri Kritzer Cabral、Marko Makela、Giuseppe Maxia、Paul McCullagh、B. Keith Murphy、Dhiren Patel、Sergey Petrunia、Alexander Rubin、Paul Tuckfield、Heikki Tuuri 和 Michael “Monty” Widenius。

有一份特别的感谢要送给 O'Reilly 的编辑 Andy Oram 和助理编辑 Isabel Kunkle, 以及审稿人 Rachel Wheeler, 同时也感谢 O'Reilly 团队里的其他成员。

来自 Baron

From Baron

我要感谢我的妻子 Lynn Rainville 和我们的狗狗 Carbo。如果你也曾写过一本书, 我确信你可体会到我有多么感谢他们。我也非常感谢 Alan Rimm-Kaufman 和我在 Rimm-Kaufman 集团的同事们, 在写书的过程中, 他们给了我支持和鼓励。我要感谢 Peter、Vadim 和 Arjen, 是他们给了我这个机会让梦想成真。最后, 我要感谢 Jeremy 和 Derek 为我们开了个好头。

来自 Peter

From Peter

我从事 MySQL 性能和伸缩性方面的讲演、培训和咨询已经很多年了，我一直想把它们扩大到更多的受众，因此，当 Andy Oram 邀请我加入本书编写中时，我感到非常兴奋。此前我没写过书，所以，我对所需要的时间和精力都毫无把握。我们先谈到只对第一版做一些更新，以跟上 MySQL 最近的版本升级，但我们想把很多新素材加到书里去，结果就几乎重写了整本书。

这本书是真正的团队合作的结晶。因为我忙于 Percona 的事情——我和 Vadim 的咨询公司，又因为英语并非我的第一语言，所以我们有不同的角色。我负责提供大纲和技术性内容，然后，我把素材都过一遍，在写作的时候再对它进行修订和扩展。当 Arjen (MySQL 文档团队的前任负责人) 加入之后，我们就开始勾划出整个提纲。在 Baron 到来后，一切才真正开动起来，他能够以不可思议的速度编写出高质量的内容。Vadim 在深入检查 MySQL 源代码和提供基准测试或其他探索来巩固我们的论点时发挥了很大的作用。

当我们编写这本书时，我们发现越来越多的领域需要刨根问底。本书主题里的大多数，例如复制、查询优化、InnoDB、架构和设计都可以分别轻易地写成一本书，因此，我们不得不在某一个阶段时停止，把余下的材料用在将来可能要出的新版上、我们的博客上、我们的讲演上，以及我们的技术文章里。

本书的评审者给予了我们巨大的帮助，无论是来自 MySQL AB 公司内部的还是外部的，他们都是这个世界上顶级的 MySQL 专家，他们包括 MySQL 的创建者 Michael Widenius、InnoDB 的创建者 Heikki Tuuri、MySQL 优化器团队的负责人 Igor Babaev，以及其他人士。

我还要感谢我的妻子 Katya Zaytseva、我的孩子 Ivan 和 Nadezhda，他们容许了我把家庭时间花在本书写作上。我也要感谢 Percona 的雇员们，当我在公司里“人间蒸发”去写书的时候，是他们处理了日常的事务。当然，我也要感谢 O'Reilly 和 Andy Oram 让这一切成为了可能。

来自 Vadim

From Vadim

我要感谢 Peter，能在本书中与他合作，我感到十分开心，期望在其他项目中能继续共事；我也要感谢 Baron，他在本书写作过程中起了很大的作用；还有 Arjen，跟他一起工作非常好玩。我还要感谢我们的编辑 Andy Oram，他抱着十二万分的耐心与我们一起工作。还要感谢 MySQL 团队，是他们创造了这个伟大的软件；我还要感谢我们的客户给予我调优 MySQL 的机会。最后，我要特别感谢我的妻子 Valerie 及我们的儿子 Myroslav 和 Timur，他们一直支持我，帮助我一步步前进。

来自 Arjen

From Arjen

我要感谢 Andy 的睿智、指导和耐心，感谢 Baron 中途加入到我们当中来，感谢 Peter 和 Vadim 坚实的背景信息和基准测试。也要感谢 Jeremy 和 Derek 在第一版里打下的基础，在我的书上，Derek 题写着：“要诚实——这就是我所有的要求。”

我也要感谢所有我在 MySQL AB 时的同事们，在那里我获得了关于本书主题的大多数知识。在此，我还要特别提到 Monty，我一直认为他是令人自豪的 MySQL 之父，尽管他的公司如今已成为 SUN 公司的一部分。我要感

谢全球 MySQL 社区里的每一个人。

最后但同样重要的是，我要感谢我的女儿 Phoebe，在她尚年少的生活舞台上，不用关心什么叫 MySQL，也不用考虑 Wiggles 所指的到底是何物。从某些方面来讲，无知就是福，它能给予我们一个全新的视角来看清生命中真正重要的是什么。对于读者，祝愿你们的书架上又增添了一本有用的书，还有，不要忘记你的生活。

本书第一版的致谢

Acknowledgments for the First Edition

像这样一本书的写成离不开许许多多人的帮助。没有他们的无私援助，你手上的这本书可能仍然是我们显示器屏幕四周的那一堆小贴纸。这是本书的一部分，在这里，我们可以感谢每一个曾经帮我们脱离困境的人，而无须担心突然奏响的背景音乐催促我们闭上嘴巴赶快走掉——如同你在电视里看到的颁奖晚会那样。

如果没有编辑 Andy Oram 坚决的督促、请求、央求和支持，我们就无法完成这个项目。如果要找出本书最负责的一个，那就是 Andy。我们真地非常感激每周一次的唠唠叨叨的会议。

其实，Andy 也不是孤独的，在 O'Reilly 里，还有一批人参与了把那些小贴纸转换成一本已装订好的你正要阅读的图书的工作，所以，我们也要感谢那些在生产、插画和销售环节的人们，感谢你们把本书合在一起。当然，还要感谢 Tim O'Reilly，是他持久不变的承诺为广大的开源软件出版了一批行业里最好的文档。

最后，我们要把感谢给予那些同意审阅本书不同阶段版本，并告诉我们哪里有错误的人们：我们的评审者。他们把 2003 年假期的一部分时间用在了审阅这些格式粗糙，充满了打字符号、误导性的语句和彻底的数学错误的文本上。我们要感谢（排名不分先后次序）：Brian “Krow” Aker、Mark “JDBC” Matthews、Jeremy “the other Jeremy” Cole、Mike “VBMySQL.com” Hillyer、Raymond “Rainman” De Roo、Jeffrey “Regex Master” Friedl、Jason DeHaan、Dan Nelson、Steve “Unix Wiz” Friedl，最后还有 Kasia “Unix Girl” Trapszo。

来自 Jeremy

From Jeremy

我要再次感谢 Andy，是他同意接纳这个项目，并持续不断鞭策我们加入更多的章节内容。Derek 的帮助非常关键，本书最后的 20%~30% 内容都是他来完成的，这使得我们不再错失下一个目标日期。感谢他同意中途加入进来，代替我只能零星爆发一下的生产力，完成了关于 XML 的繁琐工作、第 10 章、附录 C，以及我丢给他的其他那些活儿。

我也要感谢我的父母，在多年以前他们就给我买了 Commodore 64 电脑，他们不仅在前 10 年里容忍了我那如同一辈子漫长的对电子和计算机技术的沉迷，在之后还成为我不懈学习和探索的支持者。

接下来，我要感谢在过去几年里在 Yahoo! 推广 MySQL 信仰时遇到的那一群人，跟他们共事，我感到非常愉快。在本书的筹备阶段，Jeffrey Friedl 和 Ray Goldberger 给了我鼓励和反馈意见。在他们之后就是 Steve Morris、James Harvey 和 Sergey Kolychev 容忍了我在 Yahoo! Finance MySQL 服务器上做着看似固定不变的实验，即使打扰到了他们的重要工作。我也要感谢 Yahoo! 的其他成员，是他们帮我发现了 MySQL 上的那些有趣的问题和解决方法。还有，最重要地是要感谢他们对我有足够的信任和信念，让我把 MySQL 用在 Yahoo!'s 业务的重要和可见的那一部分上。

Adam Goodman，出版家和 Linux Magazine 的所有者，他帮助我轻装上阵开始为技术受众撰写文章，并在 2001 年后半年第一次出版了我的长篇 MySQL 文章。自那以后，他教授给我更多他所能认识到的关于编辑和出版的技能，还鼓励我通过在杂志上开设月度专栏在这条路上继续走下去。谢谢你，Adam。

我要感谢 Monty 和 David 与这个世界分享 MySQL。说到 MySQL AB，也要感谢在那里的其他伟大的人们，是他们鼓励我写成这本书：Kerry、Larry、Joe、Marten、Brian、Paul、Jeremy、Mark、Harrison、Matt 和团队的其他那些人。他们真的非常棒！

我要感谢我 Weblog 的所有读者，是他们鼓励我撰写基于日常工作的非正式的 MySQL 及其他技术文章。最后但同样重要的是，感谢 Goon Squad。

来自 Derek

就像 Jeremy 一样，因为太多相同的原因，我也要感谢我的家庭。我要感谢我的父母，是他们不停地鼓动我去写一本书，哪怕他们头脑中都没任何跟它相关的东西。我的祖父母给我上了两堂很有价值的课：美金的含义，以及我跟电脑相爱有多深，他们还借钱给我去购买了我平生第一台电脑：Commodore VIC-20。

我万分感谢 Jeremy 邀请我加入他那旋风般的写作过山车（bookwriting roller coaster）中来。这是一个很棒的体验，我希望将来还能跟他一起工作。

我要特别感谢 Raymond De Roo、Brian Wohlgemuth、David Calafrancesco、Tera Doty、Jay Rubin、Bill Catlan、Anthony Howe、Mark O'Neal、George Montgomery、George Barber，以及其他无数耐心听我抱怨的人，我从他们那里了解到我所努力讲述的是否能让门外汉也能理解，或者仅仅得到一个我迫切希望的笑脸。没有他们，这本书可能也会写出来，但是，我几乎可以肯定我在这过程中会疯掉。

MySQL 架构

MySQL Architecture

MySQL 架构与其他数据库服务器大不相同，这使它能够适应广泛的应用。MySQL 并非尽善尽美，但足够灵活，能适应高要求环境，例如 Web 应用。同时，MySQL 还适用于嵌入式应用、数据仓库、内容索引和分发软件、高可用的冗余系统、联机事务处理系统（OLTP）及很多其他应用类型。

为了充分发挥 MySQL 的性能，顺畅地使用它，就必须理解它的设计。MySQL 的灵活性体现在很多方面。它可以在众多硬件平台上良好地配置和运行，还支持多种数据类型。不过 MySQL 最重要、最不同寻常的特征是它的存储引擎架构，这种架构可以将查询处理（Query Processing）和各类服务器任务（Server Tasks）与数据的存储（Storage）/提取（Retrieval）相分离。在 MySQL 5.1 中，甚至支持把存储引擎作为运行时的插件（Runtime Plug-ins）动态加载。这种分离特性使用户可以基于每张表来选择存储引擎，以满足对数据存储、性能、特征及其他特性的各种需要。

本章描述了 MySQL 服务器架构的总体架构、各种存储引擎间的主要区别，以及这种区别的重要性，并试图通过简化细节和介绍示例来讨论 MySQL 的原理，这种讨论对无论是刚接触数据库服务器的新人，还是已熟悉其他数据库服务器的专家，都不无裨益。

1.1 MySQL 的逻辑架构

MySQL's Logical Architecture

如果能在头脑中构建出一幅 MySQL 各种组件如何协同工作的图像，就有助于深入理解 MySQL 服务器。图 1-1 展示了 MySQL 架构的逻辑视图。

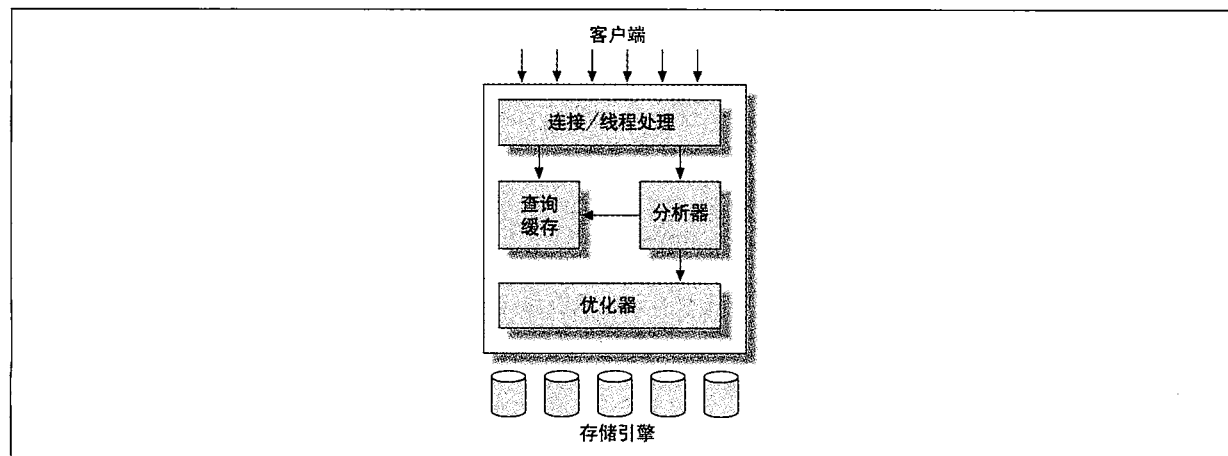


图 1-1：MySQL 服务器架构的逻辑视图

最顶层的各种服务并非 MySQL 独有。它们是一些基于网络的客户端/服务器 (Client/server) 工具或服务器都需要的服务, 比如连接处理、授权认证、安全等。

第 2 层值得关注。它包括了 MySQL 的大多数核心内容, 比如查询解析、分析、优化、缓存及所有内建函数(如日期、时间、数学和加密函数等)的代码。各种存储引擎提供的功能也集中在这层, 如存储过程、触发器、视图等。

第 3 层包含了存储引擎, 存储引擎负责存储和提取所有存放在 MySQL 中的数据。和 GNU/Linux 下的各种文件系统一样, 每个存储引擎都有自己的优势和劣势。服务器通过存储引擎 API (Storage Engine API) 与引擎进行通信。该接口隐藏了存储引擎之间的区别, 使它们在查询层上是透明的。API 包含了几十个底层函数, 用来执行相关操作, 如“开始一个事务”, “提取一行拥有某主键的数据”等等。存储引擎不会进行 SQL 解析 (注 1) (Parse), 也不会互相通信, 它们只是简单地响应服务器的请求。

1.1.1 连接管理与安全性

Connection Management and Security

每个客户连接在服务器进程中都拥有自己的线程, 每个连接所属的查询都会在指定的某个单独线程中完成, 这些线程轮流运行在某个 CPU 核心 (Core) 或 CPU 上。服务器负责缓存线程, 因此不需要为每个新的连接重建或撤销线程 (注 2)。

当客户端 (应用) 连接到 MySQL 服务器时, 服务器要对其进行认证 (Authenticate), 认证方式基于用户名、原始主机信息和口令。对于安全套接字层 (SSL) 方式的连接, 还使用了 X.509 证书。一旦某个客户端成功连接服务器, 服务器就会验证该客户端是否有权执行某个具体查询 (例如, 是否允许客户端对 World 数据库中的 Country 表执行 SELECT 语句)。第 12 章将会讨论更多相关细节。

1.1.2 优化与执行

Optimization and Execution

MySQL 会解析查询, 并创建一个内部数据结构 (解析树), 然后对其进行各种优化。其中包括重写查询, 决定查询的读表顺序, 以及选择须使用的索引等。用户可以通过特殊的关键字给优化器 (Optimizer) 传递各种提示 (Hint), 影响它的决策过程 (Decision-Making Process)。另外还可以请求服务器给出优化过程的各种说明, 使用户可以知晓服务器是如何进行优化决策的, 为用户提供一个参考基准, 方便用户重写查询、架构 (Schema) 和修改相关配置, 便于应用尽可能高效地运行。第 4 章将会讨论更多优化器的细节。

优化器并不关心某个表使用哪种存储引擎, 但存储引擎对服务器的查询优化过程有影响。优化器会请求存储引擎, 为某种具体操作提供性能与开销方面的信息, 以及表内数据的统计信息。例如, 某些存储引擎可以支持对某类查询更有利的索引类型。参见第 3 章中“索引与方案优化”一节, 了解更多相关信息。

不过, 在解析查询之前, 服务器会“询问”查询缓存, 它只能保存 SELECT 语句和相应的结果。如果能在缓存中找到将要执行的查询, 服务器就不必重新解析、优化或重新执行查询, 只须直接返回已有结果即可。在本书第 204 页“MySQL 查询缓存”一节中, 将会详细讨论相关内容。

注 1: InnoDB 是一个例外, 它会解析外键定义, 因为 MySQL 服务器自身没有实现它。

注 2: MySQL AB 计划在服务器的未来版本中分离联接和线程。

1.2 并发控制

无论何时，只要不止一个查询同时修改数据，都会产生并发控制问题。本章的目的是讨论 MySQL 在两个层面的并发控制：服务器层与存储引擎层。并发控制是一个内容庞大的主题，已有大量理论文献对其进行详细论述。本书并不想讨论相关的理论细节，也不关心 MySQL 内部的并发实现原理。本章只是给出一个概述，描述 MySQL 如何处理并发读取程序（Concurrent Reader），以及并发写入程序（Concurrent Writer），在本章其余部分，读者可以了解所需的相关内容。

以 Unix 系统的 email box 应用为例，典型的 mbox file 的格式是非常简单的。在 mbox 邮箱中，所有邮件信息都是串行的，彼此首尾相连。这使得邮件系统可以很容易读取与分析邮件信息。投递邮件也很容易，只要在文件的末尾附加新的邮件内容即可。

不过，假如两个进程在同一时间向同一邮箱投递邮件，那会发生什么情况？显然，邮箱的数据会被破坏掉，两封邮件会被交叉地保存在邮箱文件的末尾。设计良好的邮件投递系统可以使用锁定防止数据损坏。如果客户试图投递邮件，但邮箱是被锁住的，那么客户就必须等待，直到锁定解除才能投递自己的邮件。

这种方案在实际应用环境中工作良好，但它不支持并发处理。因为在任意给定时间里，只有一个进程可以修改邮箱的数据。这种方法在高容量的邮箱系统中也是个问题。

1.2.1 读锁（Read Lock）/写锁（Write Lock）

读取邮箱信息并不麻烦，即使多个用户并发读取同一邮箱，也不会有什么問題。因为读操作不会造成任何修改，所以就不会出错。不过，假如一个程序正在读邮箱，另一个用户试图删除编号 25 的邮件，那将发生什么结果？结果可能是，某一正在读的用户报错退出，或者是他看到一幅与邮箱的实际状态不符的错误视图。所以为了安全起见，即使读邮箱也必须特别注意。

可以把邮箱想象成数据库，把每封邮件想象成表中的行，就很容易发现，在这类场景里，问题都是类似的，如修改数据库表中的行，就十分类似于删除或修改邮箱文件中的邮件信息。

解决这类经典问题的方法是使用并发控制，并发控制的概念是很简单的。在处理并发读或并发写时，系统会使用一套锁系统来解决问题。这种锁系统由两类锁组成，通常称之为共享锁（Shared Lock）和排他锁（Exclusive Lock），或者叫读锁（Read Lock）和写锁（Write Lock）。

不用关心锁技术的具体实现方式，在这里描述相关锁概念如下：某一资源上的读锁是共享的，或是说是互不阻塞的。在同一时间，多个用户可以读取同一资源，而互不干扰。另一方面，写锁是排他的，也就是说，一个写锁会阻塞其他的读锁和写锁，这是出于安全策略的考虑，在给定时间里，只有一个用户能写入资源，以防止用户在写操作的同时其他用户读取同一资源。

对数据库来说，随时随地都会发生锁定。当某一用户修改某一部分数据时，MySQL 会禁止其他用户读取同一数据。大多数时候，MySQL 都是以透明的方式实现锁的内部管理。

1.2.2 锁粒度 (Lock Granularity)

图 1.2.2 锁粒度

一种提高共享资源并发性的方法就是让锁定对象更有选择性。要记住只锁定部分须修改的数据，而不是所有的资源。更理想的方式是，只对要修改的数据片精确加锁。任何时间，在给定的资源上，被加锁的数据量越小，就可以允许更多的并发修改，只要相互之间互不冲突即可。

这么做的问题是加锁也会消耗系统资源。每一种锁操作，如获得锁、检查锁是否已解除，以及释放锁等，都会增加系统的开销。如果系统花费大量时间来管理锁，而不是读/写数据，那么系统整体性能可能会因此受到影响。

所谓的锁策略，就是在锁开销和数据安全之间寻求一种平衡，这种平衡也能影响系统性能。大多数的商业数据库服务器没有提供更多的选择，通常都是在表上施加行级锁 (Row-Level Locking)，并提供种种复杂的手段，在有锁的情况下改善系统的性能。

而另一方面，MySQL 则提供了多种选择。每种 MySQL 存储引擎都可以实现独有的锁策略 (Locking Policy) 或锁粒度 (Lock Granularity)。在存储引擎设计中，锁管理 (Lock Management) 是个非常重要的议题。将锁粒度调整到某一水平，也许就能为某种应用目的提供更佳的性能，不过，这也可能使存储引擎又不适用于其他的用途了。由于 MySQL 可以提供多种存储引擎，所以它不需要一个通用解决方案。下面将介绍两种最重要的锁策略。

表锁 (Table lock)

MySQL 支持大多数基本的锁策略，其中开销最小的锁策略是表锁。表锁类似于前文所述的邮箱加锁机制：它将整个表加锁。当一个用户对表进行写操作（如插入、删除、更新）时，用户可以获得一个写锁。写锁会禁止其他任何用户的读/写操作。另外，只有无人做写操作时，用户才能获得读锁，读锁之间是互不冲突的。

在特定的环境中，表锁可能性能良好。例如，`READ LOCAL` 表锁支持某种类型的并发写操作。另外，写锁比读锁有更高的优先级，即使有读操作用户已排在队列中，一个被申请的写锁仍可以排列在锁队列的前列（写锁会被安置在读锁之前，而读锁不能排在写锁之前）。

虽然存储引擎管理自己的锁，MySQL 本身也能使用各种有效的表锁，以用于各种目的。例如，MySQL 服务器可以在语句中，如 `ALTER TABLE` 语句中，使用表锁，而不用考虑存储引擎。

行级锁 (Row locks)

行级锁可以支持最大的并发处理（同时也带来最大的锁开销）。众所周知，行级锁在 InnoDB 和 Falcon 存储引擎中已得以实现，在其他一些存储引擎也有实现。行级锁由存储引擎实现，而不是由 MySQL 服务器（如有必要，请回顾一下前文的逻辑架构图）实现。服务器完全不了解存储引擎里的锁实现方式。在本章的后续内容及全书中，所有存储引擎都是以自有方式实现加锁机制的。

1.3 事务

图 1.3.1 事务

在事务的概念产生之前的很长时间内，人们都无法利用到数据库的更多先进特征。事务是一组原子性的 SQL 查询语句，也可以被看做一个工作单元。如果数据库引擎能够成功地对数据库应用所有的查询语句，它就会执

行所有查询，但是，如果任何一条查询语句因为崩溃或其他原因而无法执行，那么所有的语句就都不会执行。也就是说，事务内的语句要么全部执行，要么一句也不执行。

本节的内容并非专属于 MySQL。如果用户已熟悉了 ACID 事务的概念，可以直接跳至本章后面，第 10 页的“MySQL 中的事务”一节。

银行应用是一个经典案例，可以解释事务应用的必要性。假设一银行数据库有两张表，checking 表和 saving 表。现在要从 Jane 的支票账户（Checking Account）里转账 200 美元到她的存款账户（Savings Account），那么需要至少完成 3 步操作：

1. 检查支票存款账户的余额是否大于 200\$。
2. 从支票存款账户余额中减去 200\$。
3. 在存款账户余额中增加 200\$。

所有的操作被打包在一个事务里，如果某一步失败，就回滚所有已完成步骤。

可以用 START TRANSACTION 语句开始一个事务，用 COMMIT 语句提交整个事务，永久地修改数据，或者用 ROLLBACK 语句回滚整个事务，取消已做的修改。事务 SQL 样本如下：

```
1  START TRANSACTION;
2  SELECT balance FROM checking WHERE customer_id = 10233276;
3  UPDATE checking SET balance = balance - 200.00 WHERE customer_id = 10233276;
4  UPDATE savings SET balance = balance + 200.00 WHERE customer_id = 10233276;
5  COMMIT;
```

单纯的事务概念不是全部的主题。试想一下，如果数据库服务器在执行第 4 条语句时突然崩溃，会发生什么事？没人知道，但用户可能会损失 200 美元。再假如，在执行第 3 和第 4 条语句之间时，另一个进程同时运行，它的目的是要先删除支票存款账户的全部余额，那么结果可能是，银行根本不知道有这个逻辑先发事件，白白给用户增加了 200 元存款。

除非系统通过 ACID 测试，否则空谈事务概念是不够的。ACID 代表了原子性 (Atomicity)、一致性 (Consistency)、隔离性 (Isolation) 和持久性 (Durability)。这些概念与事务的处理标准密切关联，一个有效的事务处理系统必须满足相关标准。

原子性 (Atomicity)

一个事务必须被视为一个单独的、内部“不可分”的工作单元，以确保整个事务要么全部执行，要么全部回滚。当一个事务具有原子性时，该事务绝对不会被部分执行，要么完全执行，要么根本不执行。

一致性 (Consistency)

数据库总是从一种一致性状态转换到另一种一致性状态。在上述例子中，一致性确保了，即使数据库系统在执行第 3、4 条语句时崩溃了，支票存款账户也不会损失 200 美元。因为最终事务根本没有被提交，任何事务处理过程中所做的数据改变，也不会影响到数据库的内容。

隔离性 (Isolation)

某个事务的结果只有在完成之后才对其他事务可见。在上述例子中，当数据库执行完第 3 条语句，还未执行第 4 条语句时，如果此时银行汇总程序也同时运行，它将仍视转账的 200 美元仍在支票存款账户内。当后文讨论隔离级时，读者就会理解为什么我们所说的通常是“不可见” (Invisible) 的。

持久性 (Durability)

一旦一个事务提交，事务所做的数据改变将是永久的。这意味着数据改变已被记录，即使系统崩溃，数据也不会因此丢失。持久性是个有点模糊的概念，因为实际上持久性也分很多级别。有些持久性策略提供一种强壮的安全保证，另一些则未必。另外，也没有什么东西是 100% 永远持久的。在本章的后续章节，将会讨论 MySQL 中持久性的真正含义，特别是在第 283 页 “InnoDB I/O 调优” 一节。

ACID 事务确保了银行不会弄丢你的钱，而这种特性在应用逻辑设计中是很难实现的，甚至不可能实现。一个 ACID 兼容的数据库服务器，要为事务处理做大量的复杂工作，确保 ACID 特性的实现，而这也许是用户未能察觉的。

正像锁粒度的增加会导致锁开销的增长一样，这种事务处理中的额外安全措施，也导致数据库服务器要完成更多的额外工作。通常，一个支持 ACID 特性的数据库，相对于不支持这种特性的数据库，需要更强的 CPU 处理能力、更大的内存和更多的磁盘空间。正如本章不断重复的，这正是选用 MySQL 存储引擎架构的有利之处。用户可以根据应用是否需要事务处理，选择相应的存储引擎。如果对于某些类型的数据查询，用户不需要真正的事务处理，他可以选择一个非事务处理型的存储引擎来实现查询，以获得更高的处理性能。用户也可以使用 LOCK TABLES 语句，为应用提供某种级别的数据保护，而这些选择完全由用户自主决定。

1.3.1 隔离级

Isolation Levels

隔离的问题比想象的要复杂。SQL 标准定义了 4 类隔离级，包括了一些具体规则，用来限定事务内外的哪些改变是可见的，哪些是不可见的。低级别的隔离级一般支持更高的并发处理，并拥有更低的系统开销。



提示：每种存储引擎实现的隔离级略有不同，如果用过其他数据库产品，用户可能发现它们未必完全满足自己的期望（因此，本节不会讨论更多的详尽细节）。读者可以根据选择的存储引擎，进一步阅读相关手册资料。

下面简单介绍四种隔离级：

READ UNCOMMITTED (读取未提交内容)

在 READ UNCOMMITTED 隔离级，所有事务都可以“看到”未提交事务的执行结果。在这种级别上，可能会产生很多问题，除非用户真的知道自己在做什么，并有很好的理由选择这样做。本隔离级很少用于实际应用，因为它的性能也不比其他级别好多少，而别的级别还有其他更多的优点。读取未提交数据，也被称之为“脏读” (Dirty Read)。

READ COMMITTED (读取提交内容)

大多数数据库系统的默认隔离级是 READ COMMITTED (但这不是 MySQL 默认的!)。它满足了隔离的早先简单定义：一个事务在开始时，只能“看见”已经提交事务所做的改变，一个事务从开始到提交前，所做的任何数据改变都是不可见的，除非已经提交。这种隔离级别也支持所谓的“不可重复读” (Nonrepeatable Read)。这意味着用户运行同一语句两次，看到的结果是不同的。

REPEATABLE READ (可重读)

REPEATABLE READ 隔离级解决了 READ UNCOMMITTED 隔离级导致的问题。它确保同一事务的多个实例在并发读取数据时，会“看到同样的”数据行。不过理论上，这会导致另一个棘手问题：幻读 (Phantom Read)。简单来说，幻读指当用户读取某一范围的数据行时，另一个事务又在该范围内插入了新行，当用户再读取

该范围的数据行时，会发现有新的“幻影”（Phantom）行。InnoDB 和 Falcon 存储引擎通过多版本并发控制（Multiversion Concurrency Control）机制解决了幻读问题。本章后面将进一步讨论这部分内容。

REPEATABLE READ 是 MySQL 的默认事务隔离级。InnoDB 和 Falcon 存储引擎都遵循这种设置，可参考第 6 章，了解如何改变这种设置。其他一些存储引擎也以此为默认设置，不过具体设置还要看相关引擎的具体规定。

SERIALIZABLE（可串行化）

SERIALIZABLE 是最高级别的隔离级，它通过强制事务排序，使之不可能相互冲突，从而解决幻读问题。简言之，SERIALIZABLE 是在每个读的数据行上加锁。在这个级别，可能导致大量的超时（Timeout）现象和锁竞争（Lock Contention）现象。作者很少看到有用户选择这种隔离级。但如果用户的应用为了数据的稳定性，需要强制减少并发的话，也可以选择这种隔离级。

表 1-1 总结了各种隔离级及相关的缺点。

表 1-1: ANSI SQL 隔离级

| 隔离级 | 脏读可能性 | 不可重复读可能性 | 幻读可能性 | 加锁读 |
|------------------|-------|----------|-------|-----|
| READ UNCOMMITTED | 是 | 是 | 是 | 否 |
| READ COMMITTED | 否 | 是 | 是 | 否 |
| REPEATABLE READ | 否 | 否 | 是 | 否 |
| SERIALIZABLE | 否 | 否 | 否 | 是 |

1.3.2 死锁

Deadlocks

死锁是指两个或多个事务在同一资源上互相占用，并请求加锁时，而导致的恶性循环现象。当多个事务以不同顺序试图加锁同一资源时，就会产生死锁。任何时间，多个事务同时加锁一个资源，一定产生死锁。例如，设想下列两个事务同时处理 StockPrice 表：

事务 1

```
START TRANSACTION;
UPDATE StockPrice SET close = 45.50 WHERE stock_id = 4 and date = '2002-05-01';
UPDATE StockPrice SET close = 19.80 WHERE stock_id = 3 and date = '2002-05-02';
COMMIT;
```

事务 2

```
START TRANSACTION;
UPDATE StockPrice SET high = 20.12 WHERE stock_id = 3 and date = '2002-05-02';
UPDATE StockPrice SET high = 47.20 WHERE stock_id = 4 and date = '2002-05-01';
COMMIT;
```

如果很不幸凑巧，每个事务在处理过程中，都执行了第一个查询，更新了数据行，也加锁了该数据行。接着，每个事务都去试图更新第二个数据行，却发现该行已被（对方）加锁，然后两个事务都开始互相等待对方完成，陷入无限等待中，除非有外部因素介入，才能解除死锁。

为了解决这种问题，数据库系统实现了各种死锁检测和死锁超时机制。对于更复杂的系统，例如 InnoDB 存储引擎，可以预知循环相关性，并立刻返回错误。这种解决方式实际很有效，否则死锁将导致很慢的查询。其他

的解决方式，是让查询达到一个锁等待超时时间，然后再放弃争用，但这种方法不够好。目前 InnoDB 处理死锁的方法是，回滚拥有最少排他行级锁的事务（一种对最易回滚事务的大致估算）。

锁现象和锁顺序是因存储引擎而异的，某些存储引擎可能会因为使用某种顺序的语句导致死锁，其他的却不会。死锁现象具有双重性：有些是因真实的数据冲突产生的，无法避免，有些则是因为存储引擎的工作方式导致的。

如果不以部分或全部的方式回滚某个事务，死锁将无法解除。在事务性的系统中，这是个无法更改的事实。用户在设计应用时，就应考虑这种问题的处理。许多应用在事务开始时，可以做简单的判定，决定重做事务。

1.3.3 事务日志

Transaction Logging

事务日志可使事务处理过程更加高效。和每次数据一改变就更新磁盘中表数据的方式不同，存储引擎可以先更新数据在内存中的拷贝。这非常快。然后，存储引擎将数据改变记录写入事务日志，它位于磁盘上，因此具有持久性。这相对较快，因为追加日志事件导致的写操作，只涉及了磁盘很小区域上的顺序 I/O (Sequential I/O)，而替代了写磁盘中表所需要的大量随机 I/O (Random I/O)。最后，相关进程会在某个时间把表数据更新到磁盘上。因此，大多存储引擎都选用了这种技术，也是通常所说的预写式日志 (Write-Ahead Logging)，利用两次磁盘写入操作把数据改变写入磁盘（注 3）。

如果数据更新已写入事务日志，却还未写入磁盘的表中，而发生系统崩溃，存储引擎将会在重启后恢复相关数据改变。具体的恢复方式因存储引擎而异。

1.3.4 MySQL 中的事务

Transactions in MySQL

MySQL AB 提供 3 个事务型存储引擎：InnoDB、NDB Cluster 和 Falcon。还有几个第三方引擎也支持事务处理，目前最知名的第三方事务性引擎是 solidDB 和 PBXT。下一节将讨论每种引擎的各自特性。

11 AUTOCOMMIT (自动提交)

MySQL 默认操作模式是 AUTOCOMMIT 模式。这意味着除非显式地开始一个事务，否则它将把每个查询视为一个单独事务自动执行。在当前连接中，可以通过变量设置，启用 (Enable) 和禁用 (Disable) AUTOCOMMIT 模式：

```
mysql> SHOW VARIABLES LIKE 'AUTOCOMMIT';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| autocommit    | ON    |
+-----+-----+
1 row in set (0.00 sec)
mysql> SET AUTOCOMMIT = 1;
```

值 1 和 ON 是等效的。同样，0 和 OFF 也是等效的。如果设置 AUTOCOMMIT=0，用户将一直处于某个事务中，直到用户执行一条 COMMIT 或 ROLLBACK 语句，之后，MySQL 将立即开始一个新事务。对于非事务型的表，如

注 3：PBXT 存储引擎聪明地避免了一些预先写入日志行为。

MyISAM 表或内存表 (Memory Table)，改变 AUTOCOMMIT 值没有意义，这些表本质上一直操作在 AUTOCOMMIT 模式。

某些命令，在一个活动事务 (Open Transaction) 中一旦执行，会在这些事务显式提交之前，直接触发 MySQL 立即提交当前事务。典型的例子，如数据定义语言 (DDL) 命令，可以导致有效的数据改变，如 ALTER TABLE 命令，另外 LOCK TABLES 和其他一些语句也会有这种效果。认真阅读当前版本的 MySQL 文档，可以获得自动提交事务的完全命令列表。

MySQL 允许使用 SET TRANSACTION ISOLATION LEVEL 命令设置隔离级，新的隔离级将在下一个事务开始后生效。用户也可以在配置文件 (参见第 6 章) 中，为整个服务器设置隔离级；或者使用下列命令，只为当前会话设置隔离级。

```
mysql> SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

MySQL 可以识别所有的 4 个 ANSI 标准隔离级，InnoDB 引擎也支持所有的隔离级。其他的存储引擎对隔离级的支持，则因不同隔离级而异。

在事务中混合使用存储引擎

MySQL 没有在服务器层管理事务，而是由下一层的存储引擎实现事务的处理。这意味着，在单一事务中，混合使用不同存储引擎并不可靠。MySQL AB 正打算为服务器增加一个高层次的事务管理服务，使用户可以安全地在事务中混合和匹配事务性表 (Transactional Table)。现在还不是时候，所以要小心一些。

在一个事务中，如果混合使用事务性表和非事务性表 (例如 InnoDB 表和 MyISAM 表)，假如事务处理一切顺利，那么结果也会正常。但是，如果事务须回滚，那么在非事务性表上做的修改将无法取消。这将导致数据库处于数据不一致的状态，在这种状态下，很难对数据进行恢复，并且事务会变得悬而未决。这正好说明了为每个表选择正常的存储引擎多么重要。

如果在一个非事务性表上进行事务性操作，MySQL 通常不会给出警告或报错信息。有时回滚事务会产生一个警告信息 “Some nontransactional changed tables couldn't be rolled back, (某些非事务性修改无法被回滚)”，但在大多数情况下，操作一个非事务性表不会得到任何提示。

隐式和显式锁定

InnoDB 使用二相锁定协议 (Two-Phase Locking Protocol)。一个事务在执行过程中的任何时候，都可以获得锁，但只有在执行 COMMIT 或 ROLLBACK 语句后，才可以释放这些锁。它会同时释放掉所有锁。前文描述的锁定机制都是隐式锁定。InnoDB 会根据用户的隔离级别，自动处理锁定。

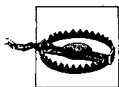
不过，InnoDB 也支持显式锁定，例如以下语句 (不代表全部)：

- SELECT ... LOCK IN SHARE MODE
- SELECT ... FOR UPDATE

MySQL 也支持 LOCK TABLES 和 UNLOCK TABLES 命令，这些命令由 MySQL 服务器实现，而不是由存储引擎。这些命令各有其用途，不能完全取代事务的作用。如果需要事务性处理，应该选择使用事务性引擎。

作者常常可以看到，一个应用从 MyISAM 引擎切换到 InnoDB 引擎后，仍然在使用 LOCK TABLES 命令，而这实际是没必要的。因为 InnoDB 已经可以实现行级加锁，况且这种命令还会引起服务器的性能问题。

12



警告：LOCK TABLES 命令与事务处理之间的交互作用，是比较复杂的。在某些服务器版本上可能会导致产生一些不可预料的行为。因此本书建议，除非是在一个事务中使用 LOCK TABLES，同时 AUTOCOMMIT 模式是被禁止的；否则，无论使用何种存储引擎，都不要使用 LOCK TABLES 命令。

1.4 多版本并发控制

大多数 MySQL 的事务性存储引擎，例如 InnoDB、Falcon 和 PBXT，不是简单地使用行加锁的机制，而是选用一种叫做多版本并发控制（MVCC，Multiversion Concurrency Control）的技术，和行加锁机制关联使用，以应对更多的并发处理问题。MVCC 不是 MySQL 独有的技术，Oracle、PostgreSQL 及其他一些数据库系统也使用同样的技术。

可以将 MVCC 设想成一种行级加锁的变形，它避免了很多情况下的加锁操作，大大降低了系统开销。依赖于具体技术实现，它可以在读取期间锁定需要的记录的同时，还允许非锁定读取。

MVCC 是通过及时保存在某些时刻的数据快照，而得以实现的。这意味着同一事务的多个实例，在同时运行时，无论每个实例运行多久，它们看到的数据视图是一致的；而同一时间，对于同一张表，不同事务看到的数据却是不同的！如果用户之前对此没有概念，这可能让人有点迷惑，但随着熟悉程度的加深，这个概念也很容易理解。

每种存储引擎实现 MVCC 的方式是不同的。例如乐观并发控制（Optimistic Concurrency Control）、悲观并发控制（Pessimistic Concurrency Control）。下面通过描述 InnoDB 简化版的行为方式，举例说明 MVCC 的工作原理。

InnoDB 通过为每个数据行增加两个隐含值的方式来实现 MVCC。这两个隐含值记录了行的创建时间，以及它的过期时间（或者叫删除时间）。每一行都存储了事件发生时的系统版本号（System Version Number），用来替代事件发生时的实际时间。每一次，开始一个新事务时，版本号都会自动递增。每个事务都会保存它在开始时的“当前系统版本”的记录，而每个查询都会根据事务的版本号，检查每行数据的版本号。下面看一下，当事务隔离级设置为 REPEATABLE READ 时，MVCC 在实际操作中的应用方式：

SELECT

InnoDB 检查每行数据，确保它们符合两个标准：

- InnoDB 只查找版本早于当前事务版本的数据行（也就是数据行的版本必须小于等于事务的版本），这确保了当前事务读取的行都是在事务开始前已经存在的，或者是由当前事务创建或修改的行。
- 数据行的删除版本必须是未定义的，或是大于事务版本的，这保证了事务读取的行，在事务开始是未被删除的。

只有通过上述两项测试的数据行，才会被当做查询结果返回。

INSERT

InnoDB 为每个新增行记录当前系统版本号。

DELETE

InnoDB 为个删除行记录当前系统版本号，作为行删除标识。

UPDATE

InnoDB 会为每个需要更新的行，建立一个新的行拷贝，并且为新的行拷贝，记录当前的系统版本号。同时，也为更新前的旧行，记录系统的版本号，作为旧行的删除版本标识。

保存这些额外记录的好处，是使大多数读操作都不必申请加锁，这使读操作变得尽可能的快，因为读操作只要选取符合标准的行数据即可。这种方式的缺点是，存储引擎必须为每行数据，存储更多的额外数据，做更多的行检查工作，以及处理一些额外的整理操作（Housekeeping Operations）。

MVCC 只工作在 REPEATABLE READ 和 READ COMMITTED 两个隔离级。READ UNCOMMITTED 隔离级不兼容 MVCC，因为在任何情况下，该隔离级下的查询，不读取符合当前事务版本的数据行，而读取最新版本的数据行。SERIALIZABLE 隔离级也不兼容 MVCC，因为该级下的读操作会对每一个返回行都进行加锁。

表 1-2 汇总了 MySQL 中的各种加锁模型和并发级别。

表 1-2：使用默认隔离级的 MySQL 锁定模型和并发性

| 加锁策略 | 并发 | 系统开销 | 引擎 |
|---------------|----|------|----------------------------|
| 表级加锁 | 最低 | 最低 | MyISAM、Merge、Memory |
| 行级加锁 | 高 | 高 | NDB Cluster |
| 支持 MVCC 的行级加锁 | 最高 | 最高 | InnoDB、Falcon、PBXT、solidDB |

1.5 MySQL 的存储引擎

MySQL's Storage Engines

本节将概述 MySQL 的存储引擎，但不会涉及过多细节，因为贯穿全书，都将一直讨论存储引擎及其相关特性。不过即使是本书，也谈不上存储引擎的完整文档；读者还应根据自己选用的存储引擎，阅读相关 MySQL 手册。MySQL 也有专门论坛讨论各种存储引擎，其中有些链接，可以经常获得更多信息和有趣的存储引擎使用方法。

如果读者只是想在更高层次上比较各种存储引擎的不同，可以直接跳至表 1-3。

在文件系统中，MySQL 会把每个数据库（也叫做架构）保存为数据目录下的一个子目录。当创建一个表时，MySQL 会在和表名同名的、以.frm 为后缀的文件中存储表的定义。例如，当创建一个名为 MyTable 的表时，MySQL 将在 MyTable.frm 中存储相关表定义。因为 MySQL 使用文件系统来存储数据库名和表定义，大小写敏感性将依赖于具体的平台。在 Windows 平台上，MySQL 的实例（Instance）名、表名、数据库名都是大小写不敏感的；在 Unix 类平台上，则是大小写敏感的。每种存储引擎对表数据和索引的存储方式有所不同，但表定义是由 MySQL 服务器独立处理的。

如须获知具体每张表使用何种存储引擎，可以用 SHOW TABLE STATUS 命令。例如，如果要检查 MySQL 数据库中的 user 表，可以执行下列命令：

```
mysql> SHOW TABLE STATUS LIKE 'user' \G
***** 1. row *****
      Name: user
      Engine: MyISAM
    Row_format: Dynamic
        Rows: 6
  Avg_row_length: 59
    Data_length: 356
Max_data_length: 4294967295
    Index_length: 2048
       Data_free: 0
Auto_increment: NULL
    Create_time: 2002-01-24 18:07:17
```

```

Update_time: 2002-01-24 21:56:29
Check_time: NULL
Collation: utf8_bin
Checksum: NULL
Create_options:
Comment: Users and global privileges
1 row in set (0.00 sec)

```

命令输出显示这是一个 MyISAM 类型表。在输出中还有其他很多信息和统计信息，下面简述每行输出的意义。

Name

表名。

Engine

表的存储引擎。在旧版本的 MySQL 中，本列的名称为 Type，而不是 Engine。

Row_format

行格式。对于 MyISAM 表，这可能是 Dynamic、Fixed 或 Compressed。动态行的行长度可变，因为它们包含有可变长的字段，例如 VARCHAR 或 BLOB 类型字段。固定行，是指行长度相同，由不可变长的字段组成，例如 CHAR 和 INTEGER 字段。压缩行只存在于压缩表中，请参见第 18 页中的“压缩的 MyISAM 表”内容。

Rows

表中的行数。对于非事务性表，这个值是精确的。对于事务性表，这通常是个估算值。

Avg_row_length

平均每行包括的字节数。

Data_length

整个表的数据量（以字节计算）。

Max_data_length

表可以容纳的最大数据量。请参见第 16 页中的“存储”一节，了解更多信息。

16 Index_length

索引数据占用磁盘空间的大小。

Data_free

对于 MyISAM 表，表示已分配，但现在未被使用的空间。这部分空间包含了以前被删除的行，这些空间可以用于以后的 INSERT 语句。

Auto_increment

下一个 AUTO_INCREMENT 值。

Create_time

表最初创建时的时间。

Update_time

表数据最近被更新的时间。

Check_time

使用 CHECK TABLE 命令或 myisamchk 工具检查表时的最近检查时间。

Collation

指本表中的默认字符集和字符列排序规则 (Collation)。请参见第 237 页“字符集与排序规则”了解更多相关信息。

Checksum

如果启用，则对整个表的内容计算实时的校验和 (Checksum)。

Create_options

指表创建时的其他所有选项。

Comment

本字段包含了其他额外信息。对于 MyISAM 表，还包含了注释，如果有注释，将是在表创建时设定的。如果表使用 InnoDB 存储引擎，将显示 InnoDB 表空间的剩余空间。如果表是个视图，注释里将包含“VIEW”的文本字样。

1.5.1 MyISAM 引擎

作为 MySQL 的默认存储引擎，在性能和可用特征之间，MyISAM 提供一种良好的平衡，这些特征包括全文检索 (Full-Text Indexing)、压缩、空间函数 (GIS)。MyISAM 不支持事务和行级锁。

存储

一般来说，MyISAM 将每个表存储成两个文件：数据文件和索引文件。两个文件的扩展名分别为 .MYD 和 .MYI。MyISAM 的格式是平台通用的，这意味着用户可以在不同架构的服务器上毫无问题地相互拷贝数据文件和索引文件。例如，可以从 Intel 架构的服务器上复制文件到 PowerPC 或 Sun SPARC 架构的服务器。

MyISAM 表可以包含动态行 (Dynamic Row) 和静态行 (Static Row，即固定长度行)。MySQL 会根据表定义决定选用何种行格式。MyISAM 表的容纳的行总数，一般只受限于数据库服务器的可用磁盘空间大小，以及操作系统允许创建的最大文件大小。

在 MySQL 5.0 中，默认配置的含有可变长行定义的 MyISAM 表可以支持 256TB 的数据处理，并使用 6 字节的指针记录数据。更早的 MySQL 版本默认使用 4 字节指针，最大可处理 4GB 数据。所有的 MySQL 版本都支持最大 8 字节的指针。如果想改变 MyISAM 表上的指针大小（调高或调低），必须在表创建选项 MAX_ROWS 和 AVG_ROW_LENGTH 中，指定相关的值，这些选项代表了用户预计使用的表大小。

```
CREATE TABLE mytable (  
  a  INTEGER NOT NULL PRIMARY KEY,  
  b  CHAR(18) NOT NULL  
) MAX_ROWS = 1000000000 AVG_ROW_LENGTH = 32;
```

在这个例子中，要求 MySQL 为表数据至少准备 32GB 的数据存储容量。可以简单地查询一下表的状态，获得 MySQL 的实际操作结果。

```
mysql> SHOW TABLE STATUS LIKE 'mytable' \G  
***** 1. row *****  
      Name: mytable  
      Engine: MyISAM  
    Row_format: Fixed  
         Rows: 0  
  Avg_row_length: 0  
    Data_length: 0
```

```
Max_data_length: 98784247807
Index_length: 1024
Data_free: 0
Auto_increment: NULL
Create_time: 2002-02-24 17:36:57
Update_time: 2002-02-24 17:36:57
Check_time: NULL
Create_options: max_rows=1000000000 avg_row_length=32
Comment:
1 row in set (0.05 sec)
```

如上如示, MySQL 精确记住了特定的表创建选项, 而且它选择的表示容量 (Representation Capable) 可以支持 91GB 的数据大小。在创建表之后, 用户也可以通过 ALTER TABLE 语句改变指针的大小, 不过这将引发整个表 and 所有相关索引的重写, 而这可能会耗费相当长的时间。

MyISAM 特性

作为 MySQL 历史最悠久的存储引擎, MyISAM 的许多特性已被开发出多年, 用于满足用户的切身需要。

18 加锁与并发

MyISAM 对整张表进行加锁, 而不是行。读取程序 (Reader) 在需要读取数据时, 在所有表上都可以获得共享锁 (读锁), 而写入程序 (Writer) 可以获得排他锁 (写锁)。用户在运行 select 查询时, 可以在同一张表内插入新行 (也称之为并发插入)。这是一个非常重要和有用的特征。

自动修复

MySQL 支持对 MyISAM 表的自动检查和自动修复。请参见 281 页 “MyISAM I/O 调优” 一节获取更多信息。

手工修复

用户可以使用 CHECK TABLE mytable 和 REPAIR TABLE mytable 命令, 检查表中的错误, 并修复错误。当服务器离线时, 也可以使用 myisamchk 命令行工具检查和修复表。

索引特性

在 MyISAM 表中, 用户可以基于 BLOB 或 TEXT 类型列的前 500 个字符, 创建相关索引。MyISAM 支持全文索引, 它可以根据个别单词, 为复杂的搜索选项创建相关索引。请参见第 3 章, 了解更多索引相关信息。

延迟更新索引 (Delayed Key Writes)

使用表创建选项 DELAY_KEY_WRITE 创建的 MyISAM 表, 在查询结束后, 不会将索引的改变数据写入磁盘, 而是在内存的键缓冲区 (In-memory Key Buffer) 中缓存索引改变数据。它只会在清理缓冲区, 或者关闭表时, 才将索引块转储到磁盘。对于数据经常改变, 并且使用频繁的表, 这种模式大大提高了表的处理性能。不过, 如果服务器或系统崩溃, 索引将肯定损坏, 并需要修复。用户可以使用脚本, 如运行 myisamchk 工具, 在重启服务器前进行修复, 也可以使用自动恢复 (Automatic Recovery) 选项进行修复 (即使没有选用 DELAY_KEY_WRITE 选项, 这些修复上的安全措施也是很有价值的)。延迟更新索引特性可以被全局配置, 也可以为个别表单独配置。

压缩的 MyISAM 表

某些表, 例如基于 CD-ROM 或 DVD-ROM 的应用, 或者某些嵌入环境下的应用, 一旦被创建和填写数据后, 数据将永不改变。这种类型的表非常适合选用压缩的 MyISAM 表 (Compressed MyISAM Table)。

使用 `myisampack` 实用工具，可以对表进行压缩（或“打包”）。压缩表一般是不能改变的（除非用户需要，也可以在解压、修改之后，再重新压缩表）。压缩表占用的磁盘空间很小，使之可以提供更快的表处理性能。因为对于查找记录，压缩后的表将减少磁盘寻道时间。压缩 MyISAM 表也可以拥有索引，但这些索引也是只读的。

对基于新一代硬件的大多数应用，读取压缩表时，因解压数据导致的系统开销并不是那么引人注目，所以这种模式的真正收益是降低了磁盘 I/O。而且，压缩表内的每一行是被单独压缩的，如果只是提取一行数据，MySQL 并不需要因此解压整个表（甚至也不需要解压一个页面）。

1.5.2 MyISAM Merge 引擎

Merge 引擎是 MyISAM 的变种。合并表（Merge Table）是指将几个相同的 MyISAM 表合并为一个虚表（Virtual Table）。它对于 MySQL 记录日志或数据仓库应用特别有用。请参见第 253 页“合并表（Merge Tables）和分区（Partitioning）”，了解更多合并表（Merge Table）信息。

1.5.3 InnoDB 引擎

InnoDB 专为事务处理设计的一款存储引擎，特别是用于处理大量短期（Short-lived）事务，短期事务是指一般能正常完成，不需要回滚的事务。InnoDB 仍然是一种最广泛应用的事务性存储引擎。它的性能水平和崩溃后自动恢复特性，也让它在非事务性存储应用中很流行。

InnoDB 将所有数据共同存储在一个或几个数据文件中，这种数据文件一般称之为表空间（Tablespace）。表空间本质上是一种“黑盒”（Black box），在“黑盒”内，InnoDB 自我管理一切数据。在 MySQL 4.1 版及更新版本中，InnoDB 也支持将每个表和相关索引存储为单独的分离文件。请参见第 290 页“InnoDB 表空间”，了解更多信息。

InnoDB 使用 MVCC 机制获取高并发性能，并且实现所有四个标准隔离级。它的默认隔离级为 REPEATABLE READ，在这个隔离级上，它使用了间隙锁（Next-key locking）策略防止“幻读”问题的产生：不仅对查询中读取的行进行加锁，而且还对索引结构中的间隙（Gaps）进行加锁，以防止幻影（Phantom）插入。

InnoDB 表是基于聚簇索引建立的，这部分内容将在第 3 章详细讨论。InnoDB 的索引结构，非常不同于其他大多数 MySQL 存储引擎，因此，它能提供一种非常快速的主键查找性能。不过，它的辅助索引（Secondary Index，也就是非主键索引）也会包含主键列，所以，如果主键定义的比较大，其他索引也将很大。如果想在表上定义很多索引，则争取尽量把主键定义得小一些。InnoDB 不会压缩索引。

直至本书写作时，InnoDB 还不能根据排序（Sorting）建立索引，而 MyISAM 支持。因此，当 InnoDB 加载数据和创建索引时，要比 MyISAM 慢很多。任何改变 InnoDB 表结构的操作，会导致整个表的重建，包括重建所有索引。

在最初设计 InnoDB 时，大多服务器仍然是低速硬盘、单 CPU 和有限的内存。如今，支持海量内存、高速硬盘的多核 CPU 服务器已经越来越便宜了。InnoDB 也面临着一些扩展性问题。

InnoDB 的开发人员正着手解决这些问题，不过在本书写作时，有些问题仍未完全解决。请参见第 296 页“InnoDB 并发调优”，以了解更多在 InnoDB 上获得高并发性能的内容。

除了高并发处理能力, InnoDB 另一个广为人知的特征是它的外键约束, 这个特征是 MySQL 服务器自身都未能提供的。当使用主键时, InnoDB 也提供了极其快速的查询查找性能。

InnoDB 做了各种内部优化。例如可预测性的预读 (Predictive read-ahead), 支持在磁盘中事先提取数据 (Prefetching); 可适应的哈希索引 (Hash index), 支持在内存中自动创建哈希索引, 可以实现极快的数据查找; 另外, 还提供了插入缓冲区, 支持快速插入。在本书的后续章节, 将详细讨论相关内容。

InnoDB 的行为是非常复杂而不易理解的, 如果读者正在使用 InnoDB, 作者强烈建议阅读 MySQL 手册中的 “InnoDB 事务模型和加锁”。当使用 InnoDB 构建应用时, InnoDB 有不少特别之处和例外, 需要事先予以了解。

1.5.4 Memory 引擎

The Memory Engine

如果用户想获得快速数据访问性能, 并且相关数据永不改变, 或者系统重启后不需要保留, 那么内存表 (Memory tables, 之前称之为 HEAP tables, 即 “堆表”) 是很用的。一般来说, 内存表要比 MyISAM 表快一个数量级。内存表的所有数据都保存在内存中, 所以相关查询根本不需要为此等待磁盘 I/O 处理。一旦系统重启, 内存表的表结构仍然会保留, 但原有的相关数据都将丢失。

内存表有几个很好的用途:

- 用于 “查找 (Lookup)” 或 “映射” 表, 例如将邮编和州名映射的表。
- 用于缓存周期性聚合数据 (Periodically Aggregated Data) 的结果。
- 用于在数据分析中产生的中间结果。

内存表支持哈希索引, 这使它在查找查询中非常快速。请参见第 101 页 “哈希索引”, 了解更多哈希索引信息。

虽然内存表非常快速, 但它还不能完全等同通常意义上的基于磁盘存储的表。内存表使用表级加锁, 只支持较低的写并发, 也不支持 TEXT 或 BLOB 字段类型。它们只支持固定大小行, 比如将 VARCHAR 类型实际存储为 CHAR 类型, 而这可能浪费不少内存。

MySQL 在处理保存中间数据的临时表时, 也会在内部使用 Memory (内存) 引擎。对于内存表而言, 如果中间数据非常大, 或者含有 TEXT 或 BLOB 字段类型, MySQL 会将其转换成 MyISAM 表, 存储在磁盘上。在后续章节中, 仍将更多地讨论相关内容。

21



提示: 人们常把内存表与临时表 (Temporary Tables) 混淆, 临时表是指使用 CREATE TEMPORARY TABLE 创建的临时性的表。临时表可以使用任何存储引擎; 它们与 Memory 引擎中使用的表不是同一个事物。临时表只在单个连接中可见, 并且当连接断开时, 也将不复存在。

1.5.5 Archive 引擎

The Archive Engine

Archive 引擎只支持 INSERT 和 SELECT 查询, 而不支持索引。Archive 引擎因为缓冲了数据写操作, 并在插入时使用 zlib 算法对每行数据进行了压缩, 因此相对于 MyISAM, 它的磁盘 I/O 消耗更少。另外, 基于此引擎的任何 SELECT 查询都会导致全表扫描 (Full Table scan)。因此, 归档表 (Archive Tables) 适合于数据记录 (Logging)

和数据采集的应用,在这种情况下,数据分析一般都要扫描全表。另外,也适合于复制服务器 (Replication Master) 上的快速数据插入。相对而言,在复制可以从服务器 (Replication Slave) 上使用其他的存储引擎,存储同样的表,这就意味着从服务器上的表可以拥有索引,在数据分析中性能会更快 (请参见第 8 章了解复制的相关内容)。

Archive 引擎支持行级加锁和专用的缓冲区系统,因而实现高并发的数据插入。在开始一个查询,并读取若干表内数据行后,Archive 引擎通过阻断 SELECT 操作,来保证读操作的一致性。另外,它也实现了批量数据的“隐形” (Invisible) 插入,直至插入过程完成。这些特点仿效了事务处理和 MVCC 的一些特征,但是 Archive 并非一个事务性存储引擎,它只是一种简单的引擎,为高速数据插入和压缩进行了优化。

1.5.6 CSV 引擎

The CSV Engine

CSV 引擎可以将“逗号分隔值 (CSV) 文件”作为表进行处理,但不支持在这种文件上建立相关索引。在服务器运行中,这种引擎支持从数据库中拷入/拷出 CSV 文件。如果从电子表格软件输出一个 CSV 文件,将其存放在 MySQL 服务器的数据 (Data) 目录中,服务器就能够马上读取相关 CSV 文件。同样,如果写数据到一个 CSV 表,外部程序也可以立刻读取它。在实现某种类型的日志记录时,CSV 表作为一种数据交换格式,特别有用。

1.5.7 Federated 引擎

The Federated Engine

Federated 引擎不在本地存储任何数据。每个联合表 (Federated Table) 都代表了一张远程 MySQL 服务器上的表,对于任何操作,它都是连接到远程服务器上完成的。有时,这可以用于一些非常规手段,例如复制时的那些技巧。

在该引擎的目前版本中,还有很多怪异之处和局限性。因为 Federated 引擎的特有工作方式,它特别适合于基于主键的单行数据查找,以及针对远程服务器的 INSERT 查询。但对于一些基本操作,如聚合查询 (Aggregate)、联接 (Join) 等,它表现得还不够好。

1.5.8 Blackhole 引擎

The Blackhole Engine

Blackhole 引擎没有任何存储机制。它会丢弃所有 INSERT 操作,而不实际存储任何数据。不过,当服务器对“黑洞表 (Blackhole Table)”进行写查询操作时,服务器会一如既往,照样记录相关日志。因此,这些操作可以被复制到从服务器上,或者只是简单地在日志中保留操作信息。这种特性让 Blackhole 引擎可以用于一种假想的复制设置,或者用于审核日志记录 (Audit Logging)。

1.5.9 NDB Cluster 引擎

The NDB Cluster Engine

2003 年,索爱公司 (Sony Ericsson) 为 MySQL AB 提供了 NDB Cluster (群集) 引擎。它最初是为满足一种高速性能需求 (实时的性能需求) 而设计的,同时支持冗余和负载均衡特性。虽然它也在磁盘上记录相关日志,但它将所有数据保存在内存中,并且优化了主键查找功能。MySQL 还为此增加了一些索引方式和许多优化方式。另外,MySQL 5.1 也支持在磁盘上存储某些字段。

NDB 的体系结构是独特的:比如,一个 NDB 群集完全不同于一个 Oracle 群集。NDB 的架构是基于无共享 (Shared-Nothing) 概念的。在这种架构中,不需要存储区域网络 (SAN) 或大型中心式存储解决方案,而其他

类型的群集可能需要这类技术。一个 NDB 数据库由数据节点 (Data node)、管理节点 (Management Node) 和 SQL 节点 (MySQL 实例) 组成。每个数据节点都保存了群集数据中的一段 (Segment) 数据。这些片段 (Fragment) 都是复制而来的, 因此在不同节点上, 系统会拥有同一数据的多份拷贝。一个物理服务器通常只专用于一个节点, 借此提供冗余特性和高可用性。从这种意义上讲, NDB 有点像服务器级的 RAID。

管理节点用于获取 (Retrieve) 集中式配置 (Centralized Configuration), 以及监控、控制所有群集节点。所有数据节点间彼此通信, 并且所有 MySQL 服务器连接所有的数据节点。对于 NDB 群集架构, 要求有低网络等待时间, 这点至关重要。

要提醒大家的是, NDB 群集是种非常“酷”的技术, 绝对值得为了满足自己的好奇心而进行探索。但是许多技术人员刻意地使用它, 把它用在本来不适用的场合。以作者的经验看, 很多人即使是非常细致地研究了这种技术, 并安装使用了一段时间, 仍然没有真正懂得这种引擎适用于什么样的环境, 以及它是如何工作的。通常的结果都是浪费时间, 因为这种引擎不是作为一种通用的存储引擎而设计的。

举一个很普通的“惊人”的例子: 目前 NDB 执行联接操作是在 MySQL 服务器层次, 而不是在存储引擎层次。因为所有 NDB 的数据都要经过网络进行检索, 复杂的联接 (Join) 操作将会因此变得非常慢。另一方面说, 单表上的查找可以非常快, 因为多个数据节点都可以提供一部分数据查找内容。而如果用户想为特定应用选择 NDB 群集, 也只是他必须深入了解的 NDB 概念中的万分之一。

23 考虑到 NDB 群集的概念如此庞大和复杂, 本书后续章节将不再讨论相关内容。如果读者对这种技术感兴趣, 应寻找专门的书籍来研究它。不过, 作者想提醒的是, 它通常不是常人理解的那种样子, 对于大多数的传统应用, 它并不适合。

1.5.10 Falcon 引擎

The Falcon Engine

Jim Starkey 是一位数据库业界的先驱, 他早先设计发明过 Interbase、MVCC 和 BOLB 字段类型, 也正是他设计了 Falcon 引擎。2006 年, MySQLAB 接纳了 Falcon 引擎, 而 Jim 目前正为 MySQLAB 工作。

Falcon 引擎是专为最新硬件所设计的, 它适合于拥有 64 位多处理器和大量内存的服务器, 也适用于更先进的硬件环境。Falcon 引擎使用 MVCC 技术, 并试图在内存中完成全部事务处理操作, 这使它在回滚操作和恢复操作方面速度非常快。

直到本书写作时, Falcon 引擎还没有完成 (例如, 它的提交操作还未能与二进制日志同步, 因此, 我们写了也不够权威。即使我们已经对它进行了初步基准测试, 但等到它投入实际应用时, 这种测试恐怕也过时了。对于大多数在线应用而言, 它表现出了很好的潜力, 不过随着时间的推移, 我们将了解更多。

1.5.11 solidDB 引擎

The solidDB Engine

solidDB 引擎是由 Solid Information Technology (<http://www.soliddb.com>) 公司开发的, 它是一款使用 MVCC 技术的事务性引擎。它同时支持悲观并发控制和乐观并发控制, 目前还没有其他的引擎这样做。solidDB for MySQL 版还包括完整的外键支持。在很多方面, 它很像 InnoDB, 例如也使用聚簇索引 (Clustered Index)。solidDB for MySQL 还包括免费的联机备份功能。

SolidDB for MySQL 产品是一个完整的产品包, 包括 solidDB 存储引擎、MyISAM 存储引擎和 MySQL 服务器。虽然 solidDB 引擎是 2006 年才与 MySQL 服务器发生联系的, 但它原有的技术与代码, 在 Solid 公司中已经成熟使用了 15 年以上。Solid 公司对整个产品提供认证和支持。这个产品的许可证方式是基于 GPL 协议的, 它也

提供双许可证模型下的商业化的许可认证，这点和 MySQL 服务器一样。

1.5.12 PBXT (Primebase XT) 引擎

Primebase XT or Primebase XT Engine

PBXT 引擎是由位于德国汉堡的 SNAP Innovation GmbH 公司 (<http://www.primebase.com>) 的 Paul McCullagh 开发的。这是一款设计独特的事务性引擎。它的一个显著特性，是使用了事务日志和数据文件来避免预写日志 (Write-ahead Logging) 操作，这大大减少了事务提交的开销。这种结构使 PBXT 引擎在处理高并发写操作时，具有潜力。并且测试也表明，在某些操作上，它比 InnoDB 更快。PBXT 使用 MVCC 技术，也支持外键约束，但不支持聚簇索引 (Clustered Index)。

24

PBXT 是一款相当新的引擎，还需要在生产环境中进一步自我证明。例如，它实现的真正持久事务 (Truly Durable Transaction) 技术，只是在作者写作本书时，最近才完成。

作为 PBXT 引擎的一项扩展，SNAP Innovation 公司还在致力研发一种可伸缩的 “blob streaming” 架构，这种技术被设计用来实现高效存储和检索大量二进制数据。

1.5.13 Maria 存储引擎

MariaDB Storage Engine

Maria 引擎是由一些 MySQL 顶级工程师研发的新型存储引擎，研发人员包括 Michael Widenius，也正是他设计了 MySQL。初始的 Maria 引擎 1.0 版只包含了设计规划中的部分特征。

Maria 引擎的目的是取代 MyISAM，——目前 MySQL 的默认存储引擎。MySQL 服务器会在执行查询时内部使用 Maria 引擎，用于创建权限表 (Privilege Table) 和临时表。目前的产品路线图中有些亮点值得关注：

- 基于每张表的事务性存储和非事务性存储的两种选项。
- 即使运行在非事务性模式下，也能进行崩溃后恢复。
- 行级锁和支持 MVCC。
- 更好的 BLOB 数据处理。

1.5.14 其他存储引擎

Other Storage Engines

不少第三方还提供了其他各式各样的引擎 (有些是具有专利的)。还有无数用于专用目的的引擎和试验性引擎 (比如，某种专用于 Web 服务查询的引擎)。在这些引擎中，有些引擎的开发是非正式的，可能只是由一两个工程师完成，这主要是因为开发一款用于 MySQL 的存储引擎，相对来说还是比较容易的。不过，大多数这类引擎还不为人知，其中部分原因也是其有限的适用性。读者可以根据自己的需要，去进一步了解相关产品。

1.5.15 选择合适的引擎

Selecting the Right Engine

在设计基于 MySQL 的应用时，要决定选择何种存储引擎来存储数据。如果不在设计阶段就考虑这个问题，在后续工作中可能会面临复杂的问题。用户可能会发现默认配置的引擎并不能提供满足自己的某种需要，比如事务。或者，应用产生的混合读写查询需要更细粒度的加锁机制，而不是 MyISAM 提供的表锁。

25

引擎的选择可以基于每一张表，因此，须要清醒考虑如何使用每张表，以及如何存储数据。这种考虑也有助于用户从整体上正确理解应用，并且预测应用的增长需求。理解和掌握相关信息，可以使用户在选择引擎时，有一个良好的判断。



提示：针对不同表选择不同的存储引擎，未必是一个上佳的方案。如果通过分析判断，可以幸运地为所有表只选择一个存储引擎，那么一般而言，这会让之后的工作变得更加容易方便。

1.5.16 考虑的因素

虽然有许多因素会影响存储引擎的选择，但通常可以归结为几个主要考虑因素。下面有一些要点是应该引起注意的。

事务

如果应用需要事务处理操作，那么在本书写作时，InnoDB 仍是最稳定的、良好集成的、已被证明的一种恰当选择。不过，随着时间的推移，作者也希望看到事务性引擎中的后起之秀，能成为 InnoDB 的有力竞争者。

如果不需要事务处理，并且主要操作是处理一些 SELECT 或 INSERT 查询，那么 MyISAM 是一个很好的选择。另外，有时应用的某个专用部分（例如日志记录）也适合选用 MyISAM 引擎。

并发

如何更好地满足并发需求，要依据应用的工作负载而定。如果只是需要并发的插入操作和读操作，那么不论读者信与不信，MyISAM 都是正确的选择！如果需要混合性的并发操作，并且操作之间要互不干扰，那么支持行级锁的引擎是更好的选择。

备份

常规备份需求也会影响表的选择。如果服务器可以周期性地停机，来进行备份，那么各种引擎都能很容易地实现备份功能。但如果需要这样或那样的某种联机备份，那么选择可能就不简单了。第 11 章有更多的内容来讨论相关的主题。

另外，要记住，如果同时使用多种引擎，会使备份和服务器（性能）调整变得更加复杂。

26 崩溃后恢复

如果有大量的数据，应该慎重考虑，系统崩溃后需要花多长时间才能恢复。比如通常来说，MyISAM 表比 InnoDB 表更容易出错，恢复所需的时间也更长。事实上，很多人并不需要事务处理，也选择了 InnoDB 引擎，就是基于这个重要的原因。

特有特性

最后，读者可能会发现应用程序依赖于只有某些存储引擎才能提供的特殊特性或优化手段。例如，很多应用都依赖于聚集索引优化。在这种情况下，只有 InnoDB 和 solidDB 可以选择。另一方面，在 MySQL 中，也只有 MyISAM 支持全文检索。如果某个存储引擎可以满足应用的一个或若干重要需求，但又不支持别的需求时，就要做折中考虑了，或者要寻找一种更灵巧的设计方案。通常的情况是，某种存储引擎可以满足所有的需要，虽然从表面上看，似乎不支持全部需求。

对于存储引擎的选择，不必马上决定。在本书其他部分，还有大量内容介绍各种引擎的优劣之处，另外还有许多有关架构与设计方面的技巧提示。总的来说，可能还有许多选项需要进一步认识考虑，等待有更多的了解后，再回头审视这个问题，可能更有裨益。

1.5.17 实例

Practical Examples

如果缺乏真实的案例，前面讨论的主题可能显得抽象。下面了解一些普通数据库应用的特点，看看各种表的特性，以及如何为这些表选择最适合的引擎。下一节将给出一个有关选项的总结。

数据记录

假设，对于来自中心电话交换机的所有电话呼叫，都要使用 MySQL 进行实时数据记录 (Logging)。或者，Apache 中已经安装了 `mod_log_sql` 模块，可以将 Web 站点的所有访问信息直接记录到表中。那么在这类应用中，速度可能是最重要的设计指标，没人希望数据库因此陷入瓶颈状态。MyISAM 引擎和 Archive 引擎非常适合这类应用，因为它们消耗的系统开销很小，并且支持每秒钟高达数千记录的插入。另外，PBXT 存储引擎可能也很适合这种数据记录应用。

事情看起来可能很简单，不过，如果还要运行报告程序，对记录的数据进行汇总，那么基于这种查询，报告程序可能要提取大量数据，而这将严重减缓数据记录的插入处理。如果出现这种情况，那该怎么办？

一种解决方案是使用 MySQL 内置的复制特性，将相关数据克隆到第二台服务器（从服务器）上，再从服务器上运行那些对时间或 CPU 要求苛刻的程序。这可以将主服务器腾出手来，继续完成数据插入。同时，从服务器上运行任何查询，都不必担心影响实时的数据记录。

27

当然，也可以选择系统在低负载时段，运行相关查询。不过随着应用规模的不断增长，这种策略是不可靠的，无法保证应用稳定运行。

另一个选择是使用合并表 (Merge Table)。调整应用，不是将所有的数据记录在同一张表里，而是将数据记录在不同表中。表名可以根据年份、原表名和月份来区分，例如名为 `Web_logs_2008_01` 或 `Web_logs_2008_jan`。然后定义一个合并表 (Merge Table)，用于相关查询，合并表中包含了须汇总的所有数据。如果要按日或按周进行汇总，策略也是一样的，只须创建含有特定名称的表即可，例如 `Web_logs_2008_01_01`。在这种设计里，汇总程序可以在那些不再须要进行写操作的表上，运行相关汇总查询；而对于当前表中，可以继续不中断地实时记录数据。

只读表 (Read-only Table) 或主读表 (Read-mostly Table)

有些表的数据用于编制目录 (Catalog) 或分列清单 (如工作岗位、竞拍物品，不动产等数据)，在这些表上的读操作远多于写操作。对于这类表，假如不介意 MyISAM 的崩溃问题，那么选择 MyISAM 是很合适的。不过不要低估崩溃问题的重要性，许多用户根本不了解使用那些不确保一定能把数据写入磁盘的存储引擎的风险。



提示：一个值得推荐的方法，是在测试服务器上模拟真实的负载，运行应用，并真的拔下电源试试崩溃。对于从崩溃中恢复数据，第一手的经验是无价之宝。这可以避免将来某一天用户因为崩溃而变得不知所措。

不要轻易相信“MyISAM 比 InnoDB 快”之类的坊间说法，这往往并不绝对正确。作者已知在很多环境下，InnoDB 都远远快过 MyISAM，特别是对于使用聚集索引，或者将数据装入内存的应用。在本书后续章节，读者可以了解更多影响存储引擎性能的各类因素（如数据大小、所需 I/O 操作的频率、主键和辅助索引等），以及这些因素对应用的影响。

订单处理

如果涉及任何订单处理，事务操作一般是必不可少的。否则，未完成的订单是无法让客户满意的。另一个需要关注的问题是，引擎是否须要支持外键约束。在本书写作时，InnoDB 可能是订单处理应用的最佳选择，虽然其他事务性引擎也是候选之一。

股票报价

如果要收集股票报价信息用于分析，那么 MyISAM 是非常适用的。当然，选择这种引擎的注意事项也是一样的。不过，如果要支持一个高流量的 Web 服务，提供实时的报价，并服务数千在线客户，那么查询过程是绝对不能容忍等待的。大量的客户可能需要同时读/写表，通常的解决方法是使用行级锁，或者在方案设计中减少更新操作的处理。

电子公告板和主题讨论论坛

对于 MySQL 用户，主题讨论论坛（Threaded Discussion Forum）应用是个有趣的问题。目前有数百个基于 PHP 或 Perl 的免费应用，支持主题讨论的功能。其中大多数应用在处理数据库操作时效率不高，比如它们常常为了每一个请求，要执行一堆查询。其中有些应用是“非数据库相关”的，所以它们的查询不能利用发挥任何一种数据库的长处。另外，这类应用还要常常更新计数器，或者为各种主题讨论计算访问统计信息。很多这样的应用只使用几个表来存储所有数据。因此，几个核心的表就变成读负载非常重，写操作也很频繁的表。另外，为了保持数据的一致性，被迫使用的加锁机制，也成为争用现象的一个重要源头。

尽管有这些设计缺陷，大多数这类应用在中低负载情况下，还是可以良好工作的。不过，如果 Web 站点的规模迅速增长，流量增加巨大，那么应用可能会因此变得非常慢。一种典型的解决方案，是换一种可以处理高负载读/写的存储引擎。不过，如果用户这么做，有时可能会惊讶地发现，应用系统比原来更慢了！

这类应用的用户们往往没意识到的问题，是应用使用了某种特殊的查询。例如下面典型的例子：

```
mysql> SELECT COUNT(*) FROM table;
```

问题是，不是所有引擎都能很快地运行这类查询：MyISAM 可以，其他的恐怕不行。对于所有引擎来说，都有类似的例子。第 2 章将帮助读者分析认识这种令人迷惑的问题，并且演示如何发现和解决这种问题，如果它存在。

CD-ROM 应用

如果要发布一个基于 CD-ROM 或 DVD-ROM，并且使用 MySQL 数据文件的应用，可以考虑使用 MyISAM 表或压缩的 MyISAM 表，这些表容易彼此隔离，并且支持在不同介质上的相互拷贝。比起非压缩的表，压缩的 MyISAM 表只占用很少的存储空间，但其数据只能是只读的。对于某些应用，这可能是个问题，但对于那些数据总放在只读介质上的特殊应用，没有理由不选择使用压缩表。

1.5.18 存储引擎总结

Storage Engines Summary

表 1-3 汇总了 MySQL 主流存储引擎与事务相关，以及与加锁相关的特点。MySQL 版本列表示了支持引擎的最低 MySQL 版本，“全部”一词是指自 MySQL 3.23 后的所有版本。

表 1-3: MySQL 存储引擎汇总

| 存储引擎 | MySQL 版本 | 事务 | 锁粒度 | 主要应用 | 忌用 |
|---------------|----------|-----|--------------|-------------------|-------------------|
| MyISAM | 全部 | 不支持 | 支持并发插入的表锁 | SELECT、INSERT、高负载 | 读写并重的场合 |
| MyISAM Merge | 全部 | 不支持 | 支持并发插入的表锁 | 分段归档、数据仓库 | 许多全局查找 |
| Memory (HEAP) | 全部 | 不支持 | 表锁 | 中间计算、静态数据查证 | 大型数据集、持久性存储 |
| InnoDB | 全部 | 支持 | 支持 MVCC 的行级锁 | 事务处理 | 无 |
| Falcon | 6.0 | 支持 | 支持 MVCC 的行级锁 | 事务处理 | 无 |
| Archive | 4.1 | 支持 | 支持 MVCC 的行级锁 | 日志记录、聚合分析 | 需要随机读取、更新、删除 |
| CSV | 4.1 | 不支持 | 表锁 | 日志记录、大规模加载外部数据 | 需要随机读取、索引 |
| Blackhole | 4.1 | 支持 | 支持 MVCC 的行级锁 | 日志记录或同步归档 | 除非有特别目的，否则不适合任何场合 |
| Federated | 5.0 | N/A | N/A | 分布式数据源 | 除非有特别目的，否则不适合任何场合 |
| NDB Cluster | 5.0 | 支持 | 行级锁 | 高可靠性 | 大部分典型应用 |
| PBXT | 5.0 | 支持 | 支持 MVCC 的行级锁 | 事务处理、日志记录 | 需要聚集索引 |
| SolidDB | 5.0 | 支持 | 支持 MVCC 的行级锁 | 事务处理 | 无 |
| Maria | 6.x | 支持 | 支持 MVCC 的行级锁 | 替代 MyISAM | 无 |

1.5.19 表转换

Table Conversion

有几种方法可以把表从一种引擎转移到另外一种引擎上，每种方法各有优缺点。下面将介绍 3 种最常用的方法。

ALTER TABLE

把表从一种引擎转移到另外一种引擎最简单的办法是使用 ALTER TABLE 语句。下列命令将表 mytable 转换成 Falcon 引擎的表：

30

```
mysql> ALTER TABLE mytable ENGINE = Falcon;
```

这种语法适合所有的存储引擎，不过这里有一个“陷阱”：这种转换过程会耗费大量时间。MySQL 为此要执行一个旧表到新表的逐行复制 (Row-By-Row Copy)。在这期间，转换操作可能会占用服务器的所有 I/O 处理能力，并且在转换时，源表要被读加锁 (Read-Locked)。因此，在一个繁忙的表上做此操作，要加以注意。作为替代手段，可以使用下一节介绍的方法，它会先做个表拷贝。

如果从一种引擎到另一种引擎做表转换，所有属于原始引擎的专用特性都会丢失。例如，如果将一个 InnoDB 表转换成 MyISAM 表，再转换回来，那么最初定义在原 InnoDB 表上的所有外键都会丢失。

转储 (Dump) 和导入 (Import)

如果想对表转换的过程做更多控制，可以选择使用 mysqldump 工具，将表先转储 (Dump) 成一个文本文件，然后再编辑转储文件 (Dump File)，修改其中的 CREATE TABLE 语句。一定要注意修改表名和引擎类型，因为即便引擎类型有所不同，同一数据库中也不允许存在同一表名的两张表。另外，mysqldump 在 CREATE TABLE 语句之前，会默认地加上 DROP TABLE 命令，如果不注意，很可能因此丢失原有数据。

请参见第 11 章了解更多有效转储和重新加载数据的方法。

CREATE 和 SELECT

第 3 种转换技术，是在第 1 种方法的速度与第 2 种方法的安全性之间做了一个平衡。它不是转储整张表，或者一次性转换所有的数据，而是创建一个新表，使用 MySQL 的 INSERT... SELECT 语法来转移数据。如下所示：

```
mysql> CREATE TABLE innodb_table LIKE myisam_table;
mysql> ALTER TABLE innodb_table ENGINE=InnoDB;
mysql> INSERT INTO innodb_table SELECT * FROM myisam_table;
```

31 如果数据量不大，这种办法效果不错。但是更高效的办法是增量地填充表，在填充每个增量数据块的时候都提交事务，这样就不会导致撤销日志 (Undo Log) 变得过于庞大。假定 id 是主键，可以重复运行下列查询（每次逐次增大 x 和 y 的值），直至所有的数据都复制到新表。

```
mysql> START TRANSACTION;
mysql> INSERT INTO innodb_table SELECT * FROM myisam_table
    -> WHERE id BETWEEN x AND y;
mysql> COMMIT;
```

转移操作完成后，源表仍会保留，可以在操作完成后删除 (Drop) 它。而此时，新表已经被填充完毕。注意：如有必要，请在转换时加锁源表，防止转移复制时数据不一致的问题。

寻找瓶颈：基准测试与性能分析

Finding Bottlenecks: Benchmarking and Profiling

有时候，你肯定需要 MySQL 提供更高的性能。但是什么是改进的对象？特定查询？数据库架构？还是硬件？解决这个问题的唯一方法是衡量系统正在进行的工作，并且测试它在各种条件下的系统性能。这也就是为什么把此章放在本书前面的原因。

最佳策略是发现并增强应用程序中的薄弱环节。如果不知道什么已经妨碍了性能，或者什么将会妨碍性能，那么这种策略就尤其有用。

基准测试 (Benchmarking) 和 **性能分析 (Profiling)** 是发现系统瓶颈的两个基本方法，它们之间有关系，但不是一回事。基准测试用来测量衡量系统的整体性能，这有助于判断系统的处理能力，揭示影响或不影响系统性能的因素；也可以用来揭示应用在处理不同数据时的性能表现。

相对来说，性能分析有助于发现应用在什么地方花费了最多时间，或者消耗了最多资源。换句话说，基准测试回答“应用或系统运行的效率如何？”而性能分析回答“为什么是这种运行效率？”

本章将分为两个部分，第一部分讨论基准测试，第二部分讨论性能分析。首先将讨论选择基准测试的原因和策略，继而讨论一些具体的基准测试方法，然后还将演示如何规划和设计基准测试，如何获取精确的测试结果，如何运行基准测试，以及如何分析相关结果，最后将简单介绍基准测试工具，以及使用这些工具的一些例子。

本章的其余部分将展示如何分析应用程序和 MySQL。我们将会使用详尽的例子来演示一些来自于实际产品的分析代码。本章还将演示如何对 MySQL 查询进行日志记录，如何分析相关日志，以及如何使用 MySQL 的状态计数器 (Status Counter) 和其他工具查明 MySQL 和查询正在进行的工作。

2.1 为什么要进行基准测试

Why Benchmark?

许多中大规模的 MySQL 部署都有专职人员负责系统基准测试。不过，所有的开发人员和 DBA 也都应该熟悉基本的基准测试原理与实施方法，因为它们有极大的用处。下面是基础测试可帮你达成的目标。

- 测试当前应用的运行状况。如果不了解当前系统的运行状况，就无法确认为此而做的调整是否有效。另外，也可以使用基准测试的结果来分析问题，这些问题往往是无法凭空预测的。
- 验证系统的扩展性。可以用基准测试模拟一个高负载状态，检测生产系统的处理能力。例如，模拟比当前多出数千倍的用户量。
- 为未来的业务增长进行规划。如果需要规划未来的负载增长，可以用基准测试来帮助估算应该使用多少

硬件，需要多大的网络容量和其他的各种相关资源。这有助于降低系统升级时的风险，也有助于降低对应用进行重大调整时的风险。

- 测试应用适应可变环境的能力。例如，可以通过基准测试发现在处理并发时，应用在随机峰值下的表现，也可以考察应用在不同系统配置下的表现，还可以用来检测应用在不同数据配置条件下的处理能力。
- 检测在不同硬件、不同软件、不同操作系统配置下的性能表现。比如，RAID5 或 RAID10，哪个更适合当前的系统？如果系统从 ATA 磁盘存储升级到 SAN 存储，是否有助于改善偶发写操作的性能？使用 2.4 版的 Linux 内核是否比 2.6 版更好？升级 MySQL 是否有助于性能提升？针对当前应用数据使用不同存储引擎，可能会有什么效果？所有这些问题都可以通过专门的基准测试获到答案。

基准测试也可以用于其他目的，例如为应用创建一个单元测试套件（Unit Test Suite）。不过，在本章只关注与性能相关的问题。

2.2 基准测试策略

Benchmarking Strategies

有两种主要的基准测试策略：将整个应用（包括 MySQL）做为整体进行测试或单独测试 MySQL 系统。这两种策略分别也被称为“集成式（Full-Stack）基准测试”和“单组件式（Single-Component）基准测试”。选择将应用做为整体来进行测试，而不是单独测试 MySQL，有下面几个理由。

- 需要测试的整个应用中，包括 Web 服务器、应用代码和数据库。测试者关心应用的整体性能，胜过关注 MySQL 自身的性能表现。
- MySQL 并非总是应用的瓶颈，集成式基准测试往往可以揭示这点。
- 只有对整个应用做测试，才能发现各个部分高速缓存的性能表现。
- 集成式基准测试可以更好揭示实际应用的真实表现，而单独测试其中某一部分很难发现这点。

另一方面，应用基准测试是很难建立的，甚至是很难设置正确的。如果基准测试的设计有问题，那么因此而做的最后决策可能也是错的，因为这样的测试结果可能没有反映出真实情况。

不过，有时候测试者可能也不需要了解整个应用的性能水平，而只关注 MySQL 的基准测试，至少在项目初期可以这样。基于有以下理由，可以选择只测试 MySQL：

- 需要比较不同架构（Schema）或查询下的系统性能表现。
- 需要针对应用中的某个具体问题进行基准测试。
- 为避免一个漫长的基准测试，需要有一个短期的基准测试，支持快速的“循环周期”，用来调整系统，并检测这种调整的效果。

另外，如果能在一个真实数据集上重复执行应用查询，那么选择针对 MySQL 的基准测试也是有用的。数据集的内容和数据大小都应真实的。如果可能，可以使用生产数据的“快照（Snapshot）”拷贝。

不幸的是，设置一个基于真实数据的基准测试可能是复杂和很耗时的。如果能得到一份生产数据集的拷贝，那当然是很幸运的。不过，这也许是不太现实的。例如，一个刚开发的新应用，只有很少的用户和应用数据。如果想了解这个应用在规模扩张后的性能表现，就只能通过模拟大量的应用数据和工作负载来测试系统。

2.2.1 测试何种指标

What to Measure



在实际测试开始前，甚至在设计基准测试之前，需要明确测试的目标。测试目标决定了实际要选用的测试工具和测试技术，决定了测试结果的精确性和意义。可以把测试目标设计为若干问题，例如“这种 CPU 是否比另一种更快？”，或“新的索引是否比当前索引性能更佳？”，等等。

也许这样的问题显得不那么清楚。不过，有时需要用不同的方法测量不同的指标，例如，针对时延（Latency）和吞吐量（Throughput）可能需要不同的测试方法。

请考虑下列测试指标，并考虑如何让它们满足自己的性能需求：

每时间单位的事务处理量

截至目前，这是一种最经典的数据库类应用基准测试指标。有关这方面的标准测试，比如 TPC-C（参见 <http://www.tpc.org>），就被广泛的引用，很多数据库厂商都试图在这类指标上得到出色的表现。这些指标是用来测试联机事务处理（OLTP）性能的，非常适用于交互式的多用户应用。常用的测试单元是每秒钟事务处理量。

术语“吞吐量”与每时间单位事务处理量（或者说每时间单位的某种工作单元量）通常含义相同。

响应时间或时延（Response Time or Latency）

这些指标用来测量任务所需的整体时间。根据具体的应用，测量的时间单位可以是毫秒、秒或分钟。根据具体的时间单位可以再确定平均响应时间、最小响应时间和最大响应时间。

通常来说，最大响应时间值意义不大。因为测试时间越长，相应的最大响应时间可能就越大。而且这也是不可重复的，每次测试结果都可能变化很大。基于这样的理由，很多人选择了百分比响应时间（Percentile Response Time）来代替最大响应时间。例如，如果 95% 响应时间是 5 毫秒，就代表了任务可以在 95% 的时间段里，在 5 毫秒内完成。

使用图表有助于理解这类测试结果。例如使用折线图（如平均值折线及 95% 百分比折线）和散点图（Scatter Plot），可以帮助测量者了解测试结果的数据分布。这些图表有助展示一种长时测试的结果表现。

假设系统每隔一小时，要为检查点（Checkpoint）操作运行耗时一分钟，那么在执行检查点操作时，系统将延迟其他操作，没有事务可以及时完成。单纯的 95% 响应时间值无法揭示这些峰值现象，这种测试结果可能掩盖问题。相反，使用图表就可以表现响应时间中的周期性峰值现象。图 2-1 演示了这点。

图 2-1 展示了每分钟事务处理量（NOTPM）。相关折线表现了明显的峰值现象，而这是总平均值（Overall Average，虚线表示的）所完全不能体现的。第 1 次峰值是因为高速缓存中缺乏缓存数据导致的，其他的峰值则是由于系统集中地刷新脏页（Dirty Page）到磁盘中导致的。如果缺乏图表展示，将很难发现这些数据偏差现象。

扩展性

对于那些在可变工作负荷下，需要继续保持性能的系统，扩展性指标十分有用。

“可变工作负荷下的性能”，是个相当抽象的概念。性能指标通常是由吞吐量或响应时间来表示的，而工作负载会随着数据库的大小的改变，发联接数的变化，或者硬件的调整而改变。

扩展性指标对容量规划十分有用，因为这可以揭示应用设计中的缺陷，而其他基准测试策略无法做到这点。例如，对于单个连接的情况，设计系统可能在响应时间方面的基准测试表现良好（这是一种很糟的基准测试策略），但当并发连接几何级数地提高后，测试结果就很差了。一种在连接大量增加后仍然试图得到稳定的响应时间的基准测试，有助于发现这类设计缺陷。

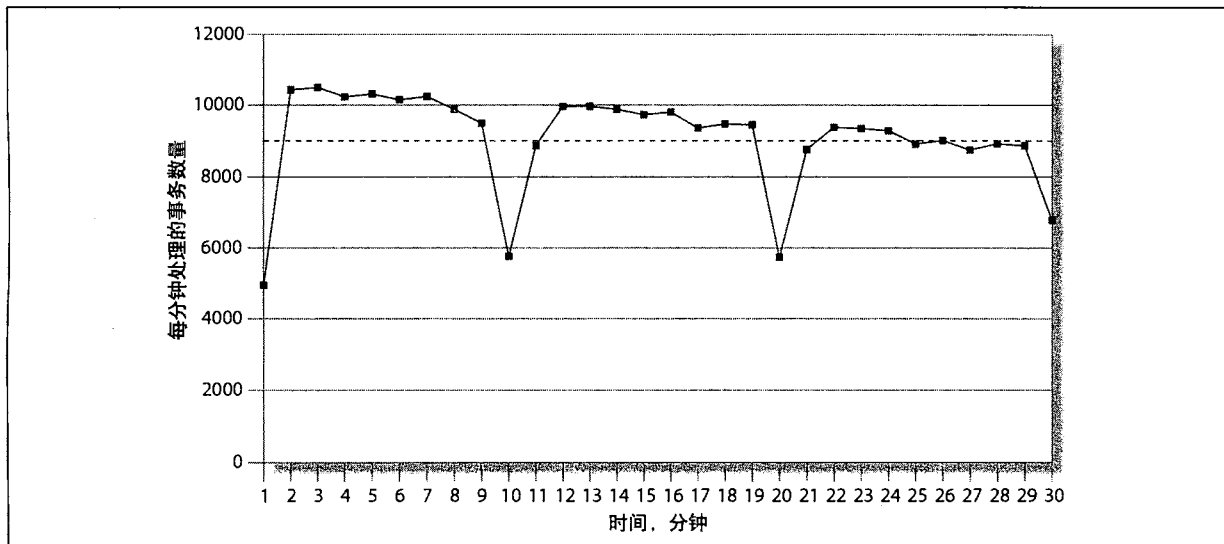


图 2-1: 运行 30 分钟 dbt2 基准测试的结果

某些作业，例如一些基于基本数据而建立汇总表 (Summary Table) 的周期性的批处理作业，也需要快速的响应时间，那么对其进行基准测试，检测响应时间指标是否满足需要，也是很有必要的。不过要记住，这类作业总是影响其他日常业务处理的。批处理作业会影响一些交互式查询的性能，反之亦然。

并发性

并发性测试是一个重要，但又常常被误解、误用的测量指标。例如，它经常被表示成同一时间有多少用户可以浏览一个 Web 站点。然而，HTTP 是一个“无状态”协议，大多用户只是简单地读取浏览器显示的信息，这不能对应成 Web 服务器的并发度。同样，Web 服务器中的并发度也未必对应成数据库服务器的并发度。这种“并发”唯一关联的是会话存储机制处理多少数据的能力。在 Web 服务器上，一个更准确的并发测量，是在峰值时间用户可以产生多少并发请求。

也可以在应用的不同环节测量相应的并发度。Web 服务器上的高并发可能会导致数据库级别的高并发，但是服务器使用的语言版本和软件工具集也会对此有影响。例如，带有连接池的 Java 应用，就比要保存永久连接的 PHP 应用，少用很多到 MySQL 服务器的并发连接。

更重要的表示方法是在给定时间里真正运行查询的连接数量。一个设计得较好的应用，程序也许会有上百个 MySQL 连接，但应该只有一部分同行运行查询。比如，一个支持 50000 用户并发访问的站点，可能只要 10 个或 15 个针对 MySQL 服务器的并发查询。

换句话说，真正需要关注的并发基准测试，是那些正在工作的并发操作，或者是同时工作的线程数量或连接数量。当并发增加时，需要测量应用性能是否降低了，如果降低了，就可能说明应用无法处理峰值负载。

需要确保应用性能不会因并发增加而下降很多，或者设计应用，使其不会产生它不能处理的高并发结果。通常来说，需要限制 MySQL 服务器上的并发度，比如使用应用排队设计。参见第 10 章以了解更多相关主题。

并发性的测量完全不同于响应时间和扩展性的测量。它不像一种结果，而更像一种如何规划设计相关测试的方法特性。应该在各种并发层次上来测试应用的性能，而不是简单地测试某个应用能达到多少并发。

归根结底，应该测试那些对于用户来说真正重要的指标。基准测试是用来测量性能的，但“性能”一词对于不同用户，有不同的含义。尽量搜集用户有关系统应该如何测量的需求（正式的或非正式的需求），比如哪种响应时间是可以接受的，什么类型的并发是要关注的，等等。然后再设计一个基准测试来满足这些需求，不要目光短浅地局限于某些指标，忽略其他的指标。

2.2.2 基准测试方法

Popular / design / article

在了解一般概念之后，现在可以讨论如何设计和执行基准测试的相关细节了。但在讨论如何使基准测试设计得更好之前，需要先看一下哪些常见错误是应该避免的，这些错误可能会导致测试结果无用，或不精确。

- 使用真实数据的一个子集，而不是全集。例如应用需要数百 GB 数据，但测试只用了 1GB 数据；或者只使用当前数据集进行测试，却希望模拟将来业务增长后的性能测试。
- 使用错误的分布式数据。例如平均地使用分布式数据，而忽略了真实系统中是有“热点区域 (Hot Spots)”的。（随机产生的数据往往不同于真实分布的。）
- 使用非真实的分布参数。例如假定所有用户的概要文件 (Profile) 都平均地被读取。
- 在多用户应用中使用单用户想定进行测试。
- 在单服务器上测试一个分布式应用。
- 与真实用户行为不匹配。例如 Web 页面中的“思考时间 (Think Time)”。真实用户在请求到一个页面后，会阅读它，而不会不停顿地一个接一个点击相关链接。
- 循环运行同一个查询。真实的查询是不相同的，而这会引起缓存内容的变化。相同的查询则会在某种程度上，全部或部分地被缓存起来。
- 忽略错误检查。如果测试出错，测试结果会毫无意义。比如一个原来很慢的操作忽然变得很快了，这就需要检查是否有错误产生。否则这种测试结果可能只是测试了 MySQL 如何快速地检查到一个 SQL 语法错误！在基准测试后，一定要检查相关错误日志，这是一个原则性的要求。
- 忽略了系统的暖机 (Warm Up) 过程，例如系统重启之后马上就进行测试。有时要了解服务器在重启多久后才能到正常的性能容量，所以要特别注意这个暖机时间 (Warm-up Period)。反过来说，如果测试的目的是想分析正常情况下的性能，那要注意，重启后马上进行基准测试时，许多缓存都还没有数据（是“冷的”，Cold），并且这种测试结果也不能反映缓存已被填满数据后的效果。
- 使用默认的服务器设置。请参见第 6 章了解更多的服务器设置优化信息。

只有避免了上述错误，才能走上改进测试品质的漫漫长路。

如果其他条件都相同，就应努力使测试过程尽可能地真实。当然，有时，稍微不太真实的测试也没关系。例如，实际应用可能运行在与数据库服务器分离的主机之上。如果使用完全相同的配置环境，会显得更加真实，但这也增加一些变数，比如网络速度、网络负载等问题。而在一个单节点上运行基准测试相对是很容易的，在某些情况下，这样的结果也足够精确。当然，这类选择需要根据实际情况进行分析做出。

2.2.3 设计和规划基准测试

Designing and Planning a Benchmark

规划基准测试的第一步是提出问题和明确目标，接着决定采用标准基准测试，还是设计专用的测试。

如果选用标准的基准测试，应确认选择适当的测试方法。例如，不要使用 TCP 方法测试一个电子商务系统。以 TCP 的解释说，“TCP 是用于测试处理海量数据的决策支持系统的”，因此，它不适合用于对 OLTP 系统进行测试。

设计自己的专用基准测试方案是复杂的，也是一个迭代（反复改进）的过程。首先，要获取一份生产数据集的快照，并确认这些数据能够恢复，可以用于后续测试。

39 然后，针对这些数据运行相关查询。可以建立一个单元测试集做为初步的基准测试，并运行多遍。不过这和真实的数据库环境还是有所差别的。更好的方法是选择一个有代表性的时间段，记录生产系统的所有查询。比如峰值时间的某一小时的查询，或者一整天的查询记录。如果记录查询的时间段选择得比较小，就需要选择多个时间段。这样有助于覆盖整个系统的活动状态，比如每周运行的报告查询，或者在非峰值时间运行的批处理作业（注 1）。

可以在不同应用级别记录相关的查询。例如，如果需要一个集成式（Full-Stack）基准测试，可以记录 Web 服务器上的 HTTP 请求，也可以开启 MySQL 的查询日志。如果需要重演这些日志，要确保重建一些分离的线程来运行查询，而不是线性地重演每个查询。对于日志中记录的每个连接，创建相互隔离的线程也是很重要的，不要只是在线程中简单地“混洗”（Shuffling）这些查询。查询日志中记录了每个查询运行需要的连接。

即使不创建自己专用的基准测试方案，也应该记录基准测试规划。基准测试将来还要被运行多次，因为需要将来能精确地复制它。应该面向未来而规划，因为下一次运行基准测试的可能就是别人了，即使仍然是你自己，你也未必能准确记住第一次的运行方案。记录的规划中应该包括测试数据、设置系统的步骤，以及“暖机”（Warm-Up）计划。

要设计一些方法来归档测试参数和测试结果，并且每次运行都要归档。归档方式可以简单地使用电子表格或记事本软件，也可以复杂点，定制设计一个数据库（请记住，可能还应该写一些脚本来帮助分析测试结果，这可以更方便、更好地处理测试结果，而不需要打开电子表格文件或文本文件）。

建立一个基准测试目录，并为每次运行建立相关的子目录，可能是很有帮助的。每次运行测试时，可以把测试结果、配置文件及相关说明都存放在适当的子目录里。如果测试的结果内容比自己需要的还多，也应把额外的测试数据也记录下来。保存一些暂不需要的数据总比丢失重要数据好，并且将来也许发现那些额外数据另有其用。在运行基准测试时，还应该记录更多的额外信息，比如 CPU 利用率、磁盘 I/O 统计信息、网络流量统计信息，以及来自 SHOW GLOBAL STATUS 命令的 counter 值，等等。

2.2.4 获得准确测试结果

Getting Accurate Results

40 获得准确测试结果的最好方法，是回答一些有关基准测试设计的基本问题：是否选择了正确的基准测试？是否为问题搜集了相关数据？是否使用了错误标准来进行基准测试？例如，是否对一个 I/O 密集型（I/O-bound）的应用，使用了 CPU 密集型的测试方法来评估性能。

注 1：当然，所有这些规定都是为了进行完美的基准测试。现实情况也是这样。

接着，确认测试结果是否可重复。要努力保证系统在每次测试前都处于相同的状态。如果是重要的测试，应该在每次测试运行之间重启系统。如果测试需要基于一个暖机（Warm Up）后的服务器，通常这是需要的，应该确认暖机过程是否足够长，是否可重复。如果暖机过程是由一系列随机查询组成的，那么测试结果就不可能重复了。

如果测试过程将改变数据或（数据库）架构（Schema），那么每次测试前，应用全新的数据快照重置系统状态。向一个包含了数千条记录的表插入数据，与向一个包含了百万条记录的表插入数据，结果是很不同的。数据存储碎片的差别及磁盘上的数据分布差别，将导致测试的结果不可重复。一个解决方法是，确保磁盘数据的物理分布一致，可以为此做一个快速格式化和分区文件拷贝。

要密切注意额外的负载（外负载，External Load），对系统保持监控性能分析，详细日志记录（Verbose Logging）、周期性的作业，和其他一些因素都会影响测试结果。一个典型的案例就是，在测试运行期间，有 cron 定时作业启动运行，或者出现周期性巡读操作（Patrol Read Cycle），或者有定期的 RAID 卡一致性检查操作。确保测试运行时，所有基准测试需要的资源都是专用于测试的。如果某些额外操作占用网络带宽，或者在一个与其他服务器共享 SAN 存储网络的环境中运行测试，那么测试结果可能是不准确的。

每次运行测试时，尽量少改变相关配置参数，用科学术语说，这叫“隔离可变因子”（Isolating The Variable）。如果必须改变某些东西，可能就要冒丢失正确结果的风险。参数之间可能是互相依赖的，很难单一地改变它们。有时甚至可能都不知道它们之间是如何关联的，这就会更增加事情的复杂性（注 2）。

（如果必须要改变参数），通常的一个解决方法是对参数进行迭代式的修改，而不是每次运行时随意地修改。例如，使用分治法（Divide-and-Conquer，也是每次运行时对分减半参数），来试图摸索服务器设置的最佳值。

作者见过很多基准测试，是用来进行预测系统迁移后的性能比较的，比如从 Oracle 迁移到 MySQL。这种测试通常是比较麻烦的，因为 MySQL 执行的查询类型与 Oracle 完全不同，它可能因此会比 Oracle 快。如果想知道基于 Oracle 的应用迁移到 MySQL 上后，性能是否会改进，通常需要为 MySQL 重新设计数据库方案（Schema）和相关查询。（在某些情况下，例如，在建立一个交叉平台应用时，可能需要知道哪些查询类型是可以运行在两种平台上的，不过这种情况也不多见。）

另外，基于默认的 MySQL 配置得到的测试结果可能没什么意义，因为这些配置都是为基于极小内存需求的极小应用而设置的。

最后，如果在测试中出现异常结果，请不要把它当成坏数据点（Data Point）简单地加以抛弃，应该认真研究，并找出产生这种结果的原因。也许会因此发现一个非常有价值的检测结果，或者一个重大的隐含问题，或者是一个测试设计中的缺陷。

2.2.5 运行基准测试和分析测试结果

Running the Benchmark and Analyzing Results

一旦准备就绪，就可以着手基准测试，收集数据和分析数据了。

通常来说，自动化基准测试过程是个好主意。这可以改进测试结果，提高测试准确度。因为自动化过程可以防止测试人员遗漏某些步骤，或者误操作。另外，它也有助于归档测试过程。

注 2：有时，这并不是问题。例如，如果正在考虑把基于 SPARC 的 Solaris 系统迁移到基于 X86 的 GNU/Linux 系统，就没有必要把在 X86 上对 Solaris 进行基准测试当成中间步骤。

可以选择任意的自动化方式，例如，可以使用 Makefile 文件，或者一组定制脚本。脚本言语可以根据自己的需要选择：shell, PHP, Perl 等。尽可能使所有的过程都能够自动化处理，包括数据加载、暖机过程、测试支行、结果记录等。



提示：一旦正确设置了自动化操作，基准测试过程将成为一步式操作。如果只是对某些应用做一次性的基准测试检查，可能没必要进行自动化处理。

基准测试通常需要运行多次。具体运行次数依赖于测试人员的计分方式 (Scoring Methodology)，以及结果的重要程度。如果想提高结果的确定性，则需要多运行几次。通常的测试实施，都是为了获得最佳结果，或者结果的平均值，或者只是运行五次，取其最佳的三次平均值。可以根据自己的需要进一步精确化测试结果。另外，也可以在测试结果上应用统计方法，比如确定置信区间 (Confidence Interval) 等，不过通常来说，不需要这种级别的确定性结果 (注 3)。只要测试过程能回答目前的需求问题，那么简单运行几遍测试，看一下结果的变化就可以了。如果结果变化很大，可以多运行几遍基准测试，或者运行的时间久一点。一般而言，这都可以减少结果的可变性。

一旦获取测试结果，就需要对其进行分析，也就是说，要将“数字”转换为“知识”，最终的目的是回答测试的需求问题。从更易理解的角度上说，可以描述为一些特定陈述，例如，“保证相同时延的情况下，升级至 4 个 CPU，可以提高 50% 的系统吞吐量”，或者“增加索引可以加快查询速度”。

如何从数据中抽象出有意义的结果，依赖于如何采集和处理数据。一般应写些脚本来分析测试结果，这不仅减少数据分析的工作量，也可以自动化整个基准测试过程：提高测试的可重复性及测试过程的归档程度。

2.3 基准测试工具

Benchmarking Tools

不用把基准测试系统搞得繁琐复杂，事实上也没这必要，除非有特别的理由，比如现有的测试工具都不可用等。目前已经有了大量测试工具可供选用。下面将介绍其中的一部分。

2.3.1 集成式测试工具

Full-Stack Tools

回忆一下前文提到的两类基准测试：集成式和单组件式基准测试。毫不奇怪，有很多工具是针对整个应用进行基准测试的，也有工具是针对 MySQL 或其他隔离组件进行压力测试的。集成式测试，通常是获得整个系统性能概观的最佳手段。已有的集成式测试工具包括：

ab

ab 是一个众所周知的 Apache HTTP 服务器基准测试工具。它可以测试 HTTP 服务器每秒钟最多可以处理多少 Web 请求。如果要测试一个 Web 应用，这个结果可以转换成整个应用每秒钟最多可以处理多少 Web 请求。ab 是个非常简单的工具，它的用途也受限于它一次只能测试一个 URL 链接。有关 ab 更多的详情可参考：<http://httpd.apache.org/docs/2.0/programs/ab.html>。

注 3：如果真地需要科学而又健壮的结果，那么就应该阅读另外的专业书籍，以学习如何设计和执行受控的测试，这已经超出了本书的范围。

http_load

这个工具在概念上与 ab 类似，它也被设计成可以加载 Web 服务器，不过它更加灵活。可以为其创建一个输入文件，文件中可包含更多不同的 URL 链接，http_load 可以在这些链接中随机选择进行测试。还可以定制 http_load，使之按定时比率（Timed Rate）测试相关请求，而不是只测试最大请求处理能力。参见 http://www.acme.com/software/http_load/ 可获取更多相关信息。

JMeter

JMeter 是一个 Java 应用，可以加载其他应用，并测试应用的性能。它可以用来测试 Web 应用，也可以用来测试 FTP 服务器，或者通过 JDBC 接口测试执行数据库查询。

JMeter 比 ab 和 http_load 复杂得多。例如，它可以通过控制一些参数，如 ramp-up time 参数，更加灵活地仿真真实用户的操作。它还有一个图形化用户接口，带有内置的图形化结果处理功能；它可以记录测试结果，并离线重演测试结果。有关 JMeter 更多详情，请参见 <http://jakarta.apache.org/jmeter/>。

2.3.2 单组件式测试工具

Database Test Suite

有一些有用的工具可以测试 MySQL 的性能和基于 MySQL 运行的系统。下一节将演示一些工具的基准测试示例。

mysqlslap

mysqlslap (<http://dev.mysql.com/doc/refman/5.1/en/mysqlslap.html>) 可以仿真服务器的负载，并报告相关计时信息。它是 MySQL 5.1 发行包的一部分，可能也能运行在 MySQL 4.1 和更新的 MySQL 版本上。可以为 mysqlslap 指定仿真的并发连接数量，并指定需要运行的 SQL 语句，SQL 语句可以是基于命令行的，也可以包含在一个文件里。如果没有指定相关 SQL 语句，mysqlslap 会自动生成用来检测服务器方案的 (Server's Schema) 的 SELECT 语句，以此进行测试。

sysbench

sysbench (<http://sysbench.sourceforge.net>) 是一个多线程系统基准测试工具。它根据各种影响数据库服务器性能的重要因子，测试系统性能。例如，它可以用来测试文件 I/O、操作系统调度程度、内存分配和传输速度、POSIX 线程，以及数据库服务器的整体性能。sysbench 支持 Lua 脚本语言 (<http://www.lua.org>)，这种语言对于测试各种类型的想定，非常方便灵活。

Database Test Suite

Database Test Suite 由开源软件开发实验室 (OSDL, Open-Source Development Labs) 设计，发布在 SourceForge 网站 (<http://sourceforge.net/projects/osdl/dbt/>) 上。Database Test Suite 是一个类似于某些工业标准测试的测试工具集，例如由事务处理性能委员会 (TPC, Transaction Processing Performance Council) 发布的各种标准。特别值得一提的是，dbt2 测试工具是一款免费的 TPC-C OLTP 测试工具 (未认证)。它支持 InnoDB 和 Falcon 存储引擎。在写作本书时，尚不清楚它是否支持其他 MySQL 存储引擎。

MySQL Benchmark Suite (sql-bench)

在 MySQL 软件发行包中，MySQL 也提供了一款自己的基准测试工具，可以用于在不同数据库服务器上进行比较测试。它是单线程的，可以用来测量服务器执行查询的速度，并报告哪种类型的操作执行得最快。

这款测试工具的主要好处，是它包含了大量预定义的测试单元，可以很方便地使用，这使它可以用来轻松

比较不同存储引擎或不同配置环境下的系统性能效率。它 also 支持高层次的测试，可以用来比较两台服务器的总体性能。另外，也可以只运行测试集中的一个子集（例如，只测试 UPDATE 语句的性能）。这些测试单元大部分都是 CPU 密集型的，但也有些短周期的测试单元需要大量的磁盘 I/O 操作。

这个工具的最大缺陷是它是单用户模式的，只使用非常小的测试用数据集，无法使用用户指定的被测站点数据进测试。另外，在同一测试的多次运行之间，测试结果可能也会变化很大。因为它是单线程模式的，且完全以串行模式运行，因此它无法评估多 CPU 配置的处理效能。但是它可以比较单 CPU 服务器的性能差别。

如果希望用它测试数据库服务器，还需要有 Perl 环境和 DBD 驱动程序支持。访问 <http://dev.mysql.com/doc/n/mysql-benchmarks.html> 可以获得相关文档资料。

Super Smack

Super Smack (<http://vegan.net/tony/supersmack/>) 是一个专用于 MySQL 和 PostgreSQL 的基准测试工具，它支持压力测试和负载生成。这是一个复杂而强大的工具，支持仿真多用户，支持加载测试数据到数据库，支持使用随机数据填充测试表。所有的测试定义都包含在“smack”文件中，这种文件可用一种简单的语言来定义客户端、表、查询和其他测试要素。

2.4 基准测试样例

2.4.1 http_load 基准测试

在本节，将展示一些上一节提到的基准测试工具的实际测试样例。这些样例未必涵盖所有测试工具，但这些样例，可以帮助读者根据自己的测试目标来选择判断，并以其做为一个入门的开端。

2.4.1 http_load

先通过一个简单样例来演示 http_load 的使用方法。使用下列相关 URL 链接，并将其定义保存在一个名为 urls.txt 的文件中。

```
http://www.mysqlperformanceblog.com/  
http://www.mysqlperformanceblog.com/page/2/  
http://www.mysqlperformanceblog.com/mysql-patches/  
http://www.mysqlperformanceblog.com/mysql-performance-presentations/  
http://www.mysqlperformanceblog.com/2006/09/06/slow-query-log-analyzes-tools/
```

使用 http_load 的最简单方法，就是循环读取访问 URL 链接。测试程序将测试最快访问速度：

```
$ http_load -parallel 1 -seconds 10 urls.txt  
19 fetches, 1 max parallel, 837929 bytes, in 10.0003 seconds  
44101.5 mean bytes/connection  
1.89995 fetches/sec, 83790.7 bytes/sec  
msecs/connect: 41.6647 mean, 56.156 max, 38.21 min  
msecs/first-response: 320.207 mean, 508.958 max, 179.308 min  
HTTP response codes:  
code 200 - 19
```

MySQL 的 BENCHMARK()函数

MySQL 有一个很方便的 BENCHMARK()函数,可以测试某种具体数据库操作的执行效率,可以为函数指定一个测试循环次数和被测试的表达式。表达式可以是任何标量表达式,例如,一个标量子查询或函数。对于比较某些操作的处理性能,这个函数是很方便的,例如可以用它来评估 MD5()函数是否比 SHA1()函数快:

```
mysql> SET @input := 'hello world';
mysql> SELECT BENCHMARK(1000000, MD5(@input));
+-----+
| BENCHMARK(1000000, MD5(@input)) |
+-----+
| 0 |
+-----+
1 row in set (2.78 sec)
mysql> SELECT BENCHMARK(1000000, SHA1(@input));
+-----+
| BENCHMARK(1000000, SHA1(@input)) |
+-----+
| 0 |
+-----+
1 row in set (3.50 sec)
```

函数的返回值永远是 0;可以用它获得客户应用在执行查询时的不同计时结果。在上面例子中,似乎 MD5()函数更快一些。然而,正确使用 BENCHMARK()函数是需要一点技巧的,需要了解它实际是如何操作的。它只是测试服务器执行表达式的速度,而不会给出任何有关查询分析或开销优化的提示。例如在上述例子中,除非表达式包括了用户变量,否则第二次测试或后续测试所执行的表达式可能是已经被缓存了(注4)。

虽然 BENCHMARK()使用很方便,但一般不使用它来完成一种实际的基准测试。因为它很难合计出真正要测试的结果。它也只能很狭隘地用于整体测试过程中的一小部分。

这个测试结果是显而易见的,只是简单的测试结果统计信息。一个稍微复杂点的测试想定,可以仿真 5 个并发用户,以循环方式读取 URL 链接,测试最快访问速度。

```
$ http_load -parallel 5 -seconds 10 urls.txt
94 fetches, 5 max parallel, 4.75565e+06 bytes, in 10.0005 seconds
50592 mean bytes/connection
9.39953 fetches/sec, 475541 bytes/sec
msecs/connect: 65.1983 mean, 169.991 max, 38.189 min
msecs/first-response: 245.014 mean, 993.059 max, 99.646 min
HTTP response codes:
code 200 - 94):
```

⚡

比较而言,还可以不测试最快访问速度,而是根据预估的请求访问比率来测试相关结果(例如按每秒 5 次请求计)。

```
$ http_load -rate 5 -seconds 10 urls.txt
48 fetches, 4 max parallel, 2.50104e+06 bytes, in 10 seconds
52105 mean bytes/connection
4.8 fetches/sec, 250104 bytes/sec
msecs/connect: 42.5931 mean, 60.462 max, 38.117 min
msecs/first-response: 246.811 mean, 546.203 max, 108.363 min
HTTP response codes:
code 200 - 48
```

注4:有人发现某个表达式执行1万次的时间和执行一次的时间相等,这其实犯了错误,因为这只是命中了缓存。通常情况下,发生这样的事情要么是缓存命中,要么就是出错了。

最后，仿真更多的负载，将访问比率提高到每秒 20 个请求。请注意，连接和响应时间会随着负载的提高而有所增加。

```
$ http_load -rate 20 -seconds 10 urls.txt
111 fetches, 89 max parallel, 5.91142e+06 bytes, in 10.0001 seconds
53256.1 mean bytes/connection
11.0998 fetches/sec, 591134 bytes/sec
msecs/connect: 100.384 mean, 211.885 max, 38.214 min
msecs/first-response: 2163.51 mean, 7862.77 max, 933.708 min
HTTP response codes:
  code 200 -- 111
```

2.4.2 sysbench

可以实现各种类型的基准测试，某种意义上它就代表了一种“完整测试”的概念。它不仅可用来测试数据库的性能，也可以测试整个系统。下面将先通过一些非 MySQL 专用的测试，来测试各个子系统的性能，这些测试可用来评估系统的整体限额（Overall Limits），而后再演示如何测试数据库的性能。

使用 sysbench 进行 CPU 基准测试

最典型的子系统测试就是 CPU 基准测试，这种测试将使用 64 位整数单位，来测试指定大小的素数计算时间。下面例子将比较两台不同服务器上的测试结果，两台服务器均为 GNU/Linux 环境。首先是第一台服务器的硬件配置：

```
[server1 ~]$ cat /proc/cpuinfo
...
model name      : AMD Opteron(tm) Processor 246
stepping        : 1
cpu MHz         : 1992.857
cache size      : 1024 KB
```

测试运行方法如下：

```
[server1 ~]$ sysbench --test=cpu --cpu-max-prime=20000 run
sysbench v0.4.8: multi-threaded system evaluation benchmark
...
Test execution summary:
  total time:                121.7404s
```

第 2 台服务器使用不同的 CPU：

```
[server2 ~]$ cat /proc/cpuinfo
...
model name      : Intel(R) Xeon(R) CPU           5130 @ 2.00GHz
stepping        : 6
cpu MHz         : 1995.005
```

测试结果如下：

```
[server1 ~]$ sysbench --test=cpu --cpu-max-prime=20000 run
sysbench v0.4.8: multi-threaded system evaluation benchmark
...
Test execution summary:
  total time:                61.8596s
```

结果简单地表明了素数计算所用时间，也很容易比较：第二台服务器的基准测试结果比第一台要快两倍。

使用 sysbench 进行文件 I/O 基准测试

fileio（文件 I/O）基准测试可以测试系统在不同类型的 I/O 负载下的处理性能，这对于比较不同的硬盘驱动器、不同的 RAID 卡和 RAID 模式，都很有帮助。可以以此来调整 I/O 子系统。

测试的第 1 步是准备测试用的数据文件，要生成足够多的超过内存限额的测试数据。如果数据填充进了内存，操作系统将会缓存其中大部分的数据，而测试结果并不能精确表示为一种 I/O 密集型的工作负荷。首先创建数据集：

```
$ sysbench --test=fileio --file-total-size=150G prepare
```

第 2 步是运行基准测试。针对不同类型的 I/O 性能，有如下一些选项：

seqwr

顺序写入。

seqrewr

顺序重写。

seqrd

顺序读取。

rndrd

随机读取。

rndwr

随机写入。

rndrw

合并的随机读/写。

下面的命令将运行文件 I/O 随机读写基准测试。

```
$ sysbench --test=fileio --file-total-size=150G --file-test-mode=rndrw  
--init-rnd=on --max-time=300 --max-requests=0 run
```

下面是结果：

```
sysbench v0.4.8: multi-threaded system evaluation benchmark
```

```
Running the test with following options:
```

```
Number of threads: 1
```

```
Initializing random number generator from timer.
```

```
Extra file open flags: 0
```

```
128 files, 1.1719Gb each
```

```
150Gb total file size
```

```
Block size 16Kb
```

```
Number of random requests for random IO: 10000
```

```
Read/Write ratio for combined random IO test: 1.50
```

```
Periodic FSYNC enabled, calling fsync( ) each 100 requests.
```

```
Calling fsync( ) at the end of test, Enabled.
```

```
Using synchronous I/O mode
```

```
Doing random r/w test
```

```
Threads started!
```

```
Time limit exceeded, exiting...
```

```
Done.
```

```
Operations performed: 40260 Read, 26840 Write, 85785 Other = 152885 Total
Read 629.06Mb Written 419.38Mb Total transferred 1.0239Gb (3.4948Mb/sec)
223.67 Requests/sec executed
```

```
Test execution summary:
total time: 300.0004s
total number of events: 67100
total time taken by event execution: 254.4601
per-request statistics:
  min: 0.0000s
  avg: 0.0038s
  max: 0.5628s
  approx. 95 percentile: 0.0099s
```

```
Threads fairness:
  events (avg/stddev): 67100.0000/0.00
  execution time (avg/stddev): 254.4601/0.00
```

输出中包含了大量信息，可用于 I/O 子系统调整的主要信息是：每秒请求的数量和总体的吞吐量。上述例子中，结果分别是每秒 223.67 个请求和每秒 3.4948 MB 的数据传输量。这些数值对于评估磁盘性能十分有用。

测试完成后，可以运行清除操作删除 sysbench 为测试创建的文件。

```
$ sysbench --test=fileio --file-total-size=150G cleanup
```

使用 sysbench 进行 OLTP 基准测试

OLTP 基准测试可以仿真事务处理的工作负荷。下面例子中将使用一个包含了百万条记录的表。第 1 步是准备测试用的表：

```
$ sysbench --test=oltp --oltp-table-size=1000000 --mysql-db=test --mysql-user=root
prepare
sysbench v0.4.8: multi-threaded system evaluation benchmark

No DB drivers specified, using mysql
Creating table 'sbtest'...
Creating 1000000 records in table 'sbtest'...
```

以上就是测试数据的准备。接着，以 8 个并发线程，只读模式，运行基准测试 60 秒钟。

```
$ sysbench --test=oltp --oltp-table-size=1000000 --mysql-db=test --mysql-user=root --
max-time=60 --oltp-read-only=on --max-requests=0 --num-threads=8 run
sysbench v0.4.8: multi-threaded system evaluation benchmark

No DB drivers specified, using mysql
WARNING: Preparing of "BEGIN" is unsupported, using emulation
(last message repeated 7 times)
Running the test with following options:
Number of threads: 8

Doing OLTP test.
Running mixed OLTP test
Doing read-only test
Using Special distribution (12 iterations, 1 pct of values are returned in 75 pct cases)
Using "BEGIN" for starting transactions
Using auto_inc on the id column
Threads started!
Time limit exceeded, exiting...
(last message repeated 7 times)
```

Done.

```
OLTP test statistics:
  queries performed:
    read:                179606
    write:               0
    other:              25658
    total:             205264
  transactions:        12829 (213.07 per sec.)
  deadlocks:           0      (0.00 per sec.)
  read/write requests: 179606 (2982.92 per sec.)
  other operations:    25658 (426.13 per sec.)
```

```
Test execution summary:
  total time:            60.2114s
  total number of events: 12829
  total time taken by event execution: 480.2086
  per-request statistics:
    min:                0.0030s
    avg:                0.0374s
    max:                1.9106s
    approx. 95 percentile: 0.1163s
```

```
Threads fairness:
  events (avg/stddev):    1603.6250/70.66
  execution time (avg/stddev): 60.0261/0.06
```

如上显示，结果中包含了相当多的信息。其中最有价值的是：

- 事务数总计（The transaction count）。
- 每秒事务处理量（The rate of transactions per second）。
- 每请求的统计信息（最小、平均、最大响应时间、95%百分比响应时间）（The per-request statistics (minimal, average, maximal, and 95th percentile time)）。
- 线程公平性（Thread-Fairness）统计信息，用来表示仿真的工作负荷如何被“公平”分配。

其他 sysbench 特性

sysbench 还可以实现其他一些基准测试，但不能用来直接测试数据库服务器的性能。

内存

测试内存的连续读写性能。

线程

测试线程调度程序的性能。这对于高负载情况下测试调度程序的处理性能特别有用。

mutex（互斥）

测试 mutex（互斥）性能。测试实现是仿真所有线程在同一时间并发运行，并为此暂时获取 mutex（互斥）锁。（mutex 是一种数据结构，用来确保排他性的操作，可以互斥地访问某些资源，也防止因并发访问而导致问题。）

seqwr

seqwr 用来测试连续写的性能，这对于测试系统的实际性能限额是很重要的。它可以用来显示 RAID 控制

器的高速缓存 (Cache) 的执行状况, 如果测试结果异常, 可视为警告信息。例如, 如果使用无电池后备的高速缓存 (Cache), 而磁盘达到每秒 3 000 次请求, 可能就有问题了, 数据可能是不安全的。

另外, 除了基准测试的模式指定参数 (--test), sysbench 还支持其他常用参数, 例如 --num-threads、--max-requests、--max-time。更多信息可查询相关文档。

2.4.3 数据库测试工具集 (Database Test Suite) 中的 dbt2 TPC-C

dbt2 TPC-C on the Database Test Suite

数据库测试工具集 (Database Test Suite) 中的 dbt2 是一款实现 TPC-C 测试的免费软件。TPC-C 是一个 TPC 组织发布的测试规范, 它可以仿真复杂的联机事务处理 (OLTP) 负载。它的测试结果中包括每分钟事务数 (TpmC), 以及每事务成本 (Price/TpmC)。这种测试结果非常依赖硬件环境, 所以公开发布的 TPC-C 测试结果中, 会包含详细的测试用服务器的配置规格。



提示: dbt2 测试并不是真正的 TPC-C。它没有得到 TPC 组织的认证, 它的测试结果与 TPC-C 结果也不具有直接的可比性。

下面看一下如何设置和运行 dbt2 基准测试的例子。这里使用 dbt2 0.37 版本, 这个版本是能用于 MySQL 的最新版本 (更新的版本包含了一些 MySQL 尚不完全支持的修正)。下面是测试的步骤。

1. 准备数据。

下列命令会在指定目录, 创建用于 10 个仓库 (Warehouse) 的测试数据。仓库总共使用约 700MB 磁盘空间。使用的磁盘空间数量与仓库的数量成正比, 因此, 用户可以调整 -w 参数, 创建满足自己需要的指定大小的数据集。

```
# src/datagen -w 10 -d /mnt/data/dbt2-w10
warehouses = 10
districts = 10
customers = 3000
items = 100000
orders = 3000
stock = 100000
new_orders = 900

Output directory of data files: /mnt/data/dbt2-w10

Generating data files for 10 warehouse(s)...
Generating item table data...
Finished item table data...
Generating warehouse table data...
Finished warehouse table data...
Generating stock table data...
```

2. 加载数据到 MySQL 数据库。

下面命令将创建一个名为 dbt2w10 的数据库, 并为其加载上一步创建的测试数据 (-d 是数据库名, -f 是测试数据所在目录)。

```
# scripts/mysql/mysql_load_db.sh -d dbt2w10 -f /mnt/data/dbt2-w10 -s /var/lib/
mysql/mysql.sock
```


3. 运行基准测试。

最后一步是运行脚本目录（Scripts Directory）中的下列命令：

```
# run_mysql.sh -c 10 -w 10 -t 300 -n dbt2w10 -u root -o /var/lib/mysql/mysql.sock
-e
*****
*                               DBT2 test for MySQL started                               *
*                                                                                       *
*               Results can be found in output/9 directory                           *
*****
*
* Test consists of 4 stages:
*
* 1. Start of client to create pool of databases connections
* 2. Start of driver to emulate terminals and transactions generation
* 3. Test
* 4. Processing of results
*
*****
DATABASE NAME:          dbt2w10
DATABASE USER:          root
DATABASE SOCKET:        /var/lib/mysql/mysql.sock
DATABASE CONNECTIONS:   10
TERMINAL THREADS:       100
SCALE FACTOR(WARHOUSES): 10
TERMINALS PER WAREHOUSE: 10
DURATION OF TEST(in sec): 300
SLEEPY in (msec) 300
ZERO DELAYS MODE:       1

Stage 1. Starting up client...
Delay for each thread - 300 msec. Will sleep for 4 sec to start 10 database
connections
CLIENT_PID = 12962

Stage 2. Starting up driver...
Delay for each thread - 300 msec. Will sleep for 34 sec to start 100 terminal
threads
All threads has spawned successfully.

Stage 3. Starting of the test. Duration of the test 300 sec

Stage 4. Processing of results...
Shutdown clients. Send TERM signal to 12962.
Response Time (s)
Transaction      %      Average : 90th %      Total      Rollbacks      %
-----
    Delivery      3.53      2.224 : 3.059      1603           0      0.00
    New Order     41.24      0.659 : 1.175     18742          172      0.92
    Order Status   3.86      0.684 : 1.228      1756           0      0.00
    Payment        39.23      0.644 : 1.161     17827           0      0.00
    Stock Level     3.59      0.652 : 1.147      1630           0      0.00
3396.95 new-order transactions per minute (NOTPM)
5.5 minute duration
0 total unknown errors
31 second(s) ramping up
```

最重要的结果是输出末尾处的这一行：

```
3396.95 new-order transactions per minute (NOTPM)
```

这里显示了每分钟系统可以处理的事务数，这个数字越大越好。（“new-order”不是事务类型的专用术语，它只是表明测试是模仿客户在假想的电子商务网站下的定单。）

修改某些参数可以创建不同的基准测试。

- c 至数据库的连接数量。修改此参数可以仿真不同等级的并发水平，并观察系统测试结果的变化。
- e 此参数使用零延迟（Zero-Delay）模式，这意味着在不同查询之间将没有时间延迟。这是一个对数据库的压力测试，但并不切合实际，因为真实的用户在产生一个新查询前总需要一个“思考时间”。
- t 基准测试的总体持续时间（有效期）。选择时间时应精心设置，否则测试结果可能是无意义的。对于 I/O 密集型的工作负荷进行基准测试，太短的时间可能会导致错误的结果，因为系统可能还没有足够的时间对高速缓冲（Cache）进行“暖机”，从而正常工作。另一方面讲，如果要对 CPU 密集型的工作负荷进行基准测试，这个时间也应设置得长一些，否则数据集可能显著增大，而成为 I/O 密集型的测试。

这种基准测试的结果，可以比单纯的性能测试提供更多的信息，例如，如果发现有很多回滚（Rollback）现象，可以判定某些地方可能出错了。

2.4.4 MySQL 基准测试集（MySQL Benchmark Suite）

MySQL Benchmark Suite

MySQL 基准测试集包含了一组基于 Perl 开发的基准测试工具，在运行它们时，需要有 Perl 环境。这些测试工具位于 MySQL 安装目录的 sql-bench/ 子目录。例如，在 Debian GNU/Linux 系统上，工具位于 /usr/share/mysql/sql-bench/ 目录。

在运行命令前，应该阅读一下其中的 README 文件，这文件中包含了工具使用方法和命令行参数说明。如果要运行全部测试，可以使用下列命令。

```
$ cd /usr/share/mysql/sql-bench/
sql-bench$ ./run-all-tests --server=mysql --user=root --log --fast
Test finished. You can find the result in:
output/RUN-mysql_fast-Linux_2.4.18_686_smp_i686
```

全部基准测试运行可能会花费相当长的时间，——可能超过一小时，时间长短依赖于硬件环境和测试配置。

如果在命令行中加--log 选项，可以在测试运行期间监测测试运行状态。每项测试的结果会存放在名为 output 的子目录里。在每项测试的结果文件中，都会包含一系列的操作计时信息。下面有一个样例。为方便印刷，结果的样式稍做了调整。

```
sql-bench$ tail -5 output/select-mysql_fast-Linux_2.4.18_686_smp_i686
Time for count_distinct_group_on_key (1000:6000):
 34 wallclock secs ( 0.20 usr 0.08 sys + 0.00 cusr 0.00 csys = 0.28 CPU)
Time for count_distinct_group_on_key_parts (1000:100000):
 34 wallclock secs ( 0.57 usr 0.27 sys + 0.00 cusr 0.00 csys = 0.84 CPU)
Time for count_distinct_group (1000:100000):
 34 wallclock secs ( 0.59 usr 0.20 sys + 0.00 cusr 0.00 csys = 0.79 CPU)
Time for count_distinct_big (100:1000000):
 8 wallclock secs ( 4.22 usr 2.20 sys + 0.00 cusr 0.00 csys = 6.42 CPU)
Total time:
868 wallclock secs (33.24 usr 9.55 sys + 0.00 cusr 0.00 csys = 42.79 CPU)
```

如上所示，count_distinct_group_on_key (1000:6000)测试花费了 34 秒挂钟（Wall-Clock）时间。这是客户端运行测试花费的总体时间。其他值（Values）（usr、sys、cusr、csys）则为运行测试附加了 0.28 秒的开销。

这是运行测试客户端代码所花费的时间，而不是 MySQL 服务器的等待响应时间。这意味着测试者真正关心的结果，是除去客户端控制的部分，即实际运行时间应该是 33.72 秒。

除了运行全部测试集，也可以选择单独运行其中部分测试。例如，可以选择只运行 insert 测试，这类测试会比运行全部测试集得到的汇总信息给出更多的细节信息：

```
sql-bench$ ./test-insert
Testing server 'MySQL 4.0.13 log' at 2003-05-18 11:02:39

Testing the speed of inserting data into 1 table and do some selects on it.
The tests are done with a table that has 100000 rows.

Generating random keys
Creating tables
Inserting 100000 rows in order
Inserting 100000 rows in reverse order
Inserting 100000 rows in random order
Time for insert (300000):
  42 wallclock secs ( 7.91 usr  5.03 sys +  0.00 cusr  0.00 csys = 12.94 CPU)
Testing insert of duplicates
Time for insert_duplicates (100000):
  16 wallclock secs ( 2.28 usr  1.89 sys +  0.00 cusr  0.00 csys =  4.17 CPU)
```

2.5 性能分析 (Profiling)

Profiling

性能分析 (Profiling) 用来揭示系统的各个组成部分对系统整体运行所造成的开销影响。

最简单的开销度量值是时间，不过性能分析还能测量函数调用的数量、I/O 操作的数量、数据库查询的数量，等等。性能分析的目的是帮助用户理解系统为何是这样的运行状况。

2.5.1 对应用进行性能分析

Profiling and Application

就像基准测试一样，性能分析也可以分为应用级别和单组件级别，例如 MySQL 服务器级别。应用级别的性能分析通常关注如何优化应用，并为此提供更多精确分析结果，因为这些分析结果包含了应用系统的整体表现水平。例如，如果用户关心应用中 MySQL 查询方面的优化，那么可能只会运行或分析相关查询。而如果仅这样做，可能会失去与查询相关的其他很多重要信息，例如一些整体应用中不得不做的一些操作，如读取查询结果到内存，处理相关结果等（注 5）。

由于 Web 应用是 MySQL 应用中最常见的应用类型，这里使用一个 PHP Web 站点来演示样例。通常用户主要通过分析应用整体性能来了解系统的负载情况，但也可能需要隔离某些感兴趣的子系统，进行单独分析，例如搜索函数（The Search Function）。任何开销过大的子系统都适合进行隔离分析。

在优化一个使用 MySQL 的 PHP Web 站点时，需要基于 PHP 代码中的对象（Objects，或者模块（Moudules））

注 5：如果正在调查瓶颈，就可以通过检查一些基本的操作系统统计值了解发生的事情，这是一种捷径。如果 Web 服务器处于空闲状态，并且 MySQL 的 CPU 使用率为 100%，就不需要分析整个应用程序，尤其是服务器处于危机状态时。可以在解除危机状态后检查整个应用程序。

粒度 (Granularity)，来收集相关统计信息。分析的目标是测试每个页面在数据库操作中消耗的响应时间，数据库访问效能往往是，但不总是应用的性能瓶颈。性能瓶颈也可能是下列因素产生的：

- 外部资源，如 Web 服务或搜索引擎。
- 一些操作需要处理大量的应用数据，例如解析 (Parsing) 大型 XML 文件。
- 一些开销大的操作使用了频繁的循环处理，例如滥用正则表达式 (Regular Expressions)。
- 糟糕的优化算法，例如使用简单的在列表中查找项目的搜索算法。

在关注 MySQL 查询问题之前，就应该对应用性能的问题源头有所判断。应用级性能分析有助于发现瓶颈，这是监控和改进应用整体性能的重要一步。

分析对象及如何分析

对于大多数应用，时间是主要的性能分析指标，因为最终用户大多很介意时延问题。在 Web 应用中，可以使用 debug 模式，显示每个页面的查询，以及响应时间、记录数量。在一些慢查询操作中，也可以运行 `EXPLAIN` 来进行分析（在后续章节可以获得有关 `EXPLAIN` 的信息）。至于更一步的分析，则需要与 MySQL 服务器的一些分析指标结合处理了。

建议在新项目开始之初，就在应用中加入性能分析代码。在一个已有的应用中加入性能分析代码可能是比较困难的，但对于新应用，一般是很容易的。许多软件库 (Library) 都支持这种特征，且使用方便。例如，Java 的 JDBC 的 PHP 的 `mysqli database access libraries` 库就内嵌了这种性，支持对数据库访问的性能分析。

性能分析代码对于跟踪只在生产系统中出现的奇怪问题也是非常有价值的，这些问题往往很难在开发环境中重现。

应用的性能分析代码至少应该收集和记录下列信息：

- 总体执行时间，或者叫“挂钟时间 (Wall-Clock Time)”（对于 Web 应用，这是页面总体访问时间）。
- 每一次查询，以及其执行时间。
- MySQL 服务器打开的每一个连接。
- 对外部资源的每次调用，例如 Web 服务、内存高速缓存、调用的外部脚本等。
- 潜在的高开销函数调用，例如 XML 解析函数。
- 用户时间和系统 CPU 时间。

这些信息可以使监控性能的工作变得更加容易，它有助于深入了解性能方面的各种问题，而这是其他手段可能难以做到的，例如：

- 全局性能问题。
- 偶发的响应时间增加问题。
- 系统瓶颈，特别是与 MySQL 无关的。
- “不可见” (Invisible) 用户的执行时间，例如来自搜索引擎蜘蛛 (Spider) 的操作。

一个 PHP 性能分析样例

为了证明对 PHP Web 应用进行性能分析是多么容易简单, 来看一些代码的例子。第一个例子演示了如何配置应用, 在 MySQL 日志表 (Log Table) 中, 记录查询日志和其他性能分析数据, 并分析数据结果。

为了减少记录分析日志的负面影响, 一般在内存中提取日志信息, 只有当整个页面执行完毕才会写一条日志记录。这种方法比记录所有查询的日志要好, 因为如果记录每条访问 MySQL 的查询, 将会使查询的总量加倍。详细记录性能分析的所有数据, 实际上也很难发现性能瓶颈问题, 因为数据粒度太小, 很难标识和分析应用的故障所在。

性能分析本身会影响系统性能吗?

是的, 性能分析和常规的监控都会增加系统开销。问题是, 它们会增加多少开销, 以及由此获得的收益是否值得这种开销。

大多数用户在设计 and 搭建高性能应用时, 认为应该测试所有能够测试的内容, 但只接受这种测试只会增加系统部分开销观点。既使用户难以认同, 在应用系统中配置一些永久性的轻量级的性能分析功能, 也是非常值得的。如果仅仅因为搭建的系统不能捕捉每天的性能变化, 而产生性能瓶颈, 就非常让人不快了。同样, 如果想要发现性能问题, 历史数据的作用是无价的。另外, 性能分析数据也可以有助于硬件采购、配置资源、预测峰值时间、峰值季节负载。

什么是所谓的“轻量级”性能分析? 为所有 SQL 查询计时, 附上测量脚本的总体执行时间, 在系统开销上并不昂贵, 并且不必为所有页面的浏览做如此设置。如果确保拥有足够的流量值, 只在应用设置文件 (Setup File) 中允许性能分析功能, 就可以对一个随机样本进行性能分析。

```
<?php
$profiling_enabled = rand(0, 100) > 99;
?>
```

只对页面的 1% 浏览量进行性能分析, 就可以有助于发现最严重的问题。

在运行基准测试时, 要确保日志记录、性能分析和性能测量所占的开销被分别评估, 否则它会影响基准测试结果的准确性。

下面有一个使用了基本的 PHP 5 日志类——class.Timer.php 的简化示例, 它使用了内建的函数, 比如 getrusage() 来确定脚本的资源使用情况。

```
1  <?php
2  /*
3   * Class Timer, implementation of time logging in PHP
4   */
5
6  class Timer {
7      private $aTIMES = array( );
8
9      function startTime($point)
10     {
11         $dat = getrusage( );
12
13         $this->aTIMES[$point]['start'] = microtime(TRUE);
14         $this->aTIMES[$point]['start_utime'] =
15             $dat["ru_utime.tv_sec"]*1e6+$dat["ru_utime.tv_usec"];
16         $this->aTIMES[$point]['start_stime'] =
```

```

17         $dat["ru_stime.tv_sec"]*1e6+$dat["ru_stime.tv_usec"];
18     }
20     function stopTime($point, $comment='')
21     {
22         $dat = getrusage( );
23         $this->aTIMES[$point]['end'] = microtime(TRUE);
24         $this->aTIMES[$point]['end_utime'] =
25             $dat["ru_utime.tv_sec"] * 1e6 + $dat["ru_utime.tv_usec"];
26         $this->aTIMES[$point]['end_stime'] =
27             $dat["ru_stime.tv_sec"] * 1e6 + $dat["ru_stime.tv_usec"];
29         $this->aTIMES[$point]['comment'] .= $comment;
31         $this->aTIMES[$point]['sum'] +=
32             $this->aTIMES[$point]['end'] - $this->aTIMES[$point]['start'];
33         $this->aTIMES[$point]['sum_utime'] +=
34             ($this->aTIMES[$point]['end_utime'] -
35              $this->aTIMES[$point]['start_utime']) / 1e6;
36         $this->aTIMES[$point]['sum_stime'] +=
37             ($this->aTIMES[$point]['end_stime'] -
38              $this->aTIMES[$point]['start_stime']) / 1e6;
39     }
41     function logdata( ) {
43         $query_logger = DBQueryLog::getInstance('DBQueryLog');
44         $data['utime'] = $this->aTIMES['Page']['sum_utime'];
45         $data['wtime'] = $this->aTIMES['Page']['sum'];
46         $data['stime'] = $this->aTIMES['Page']['sum_stime'];
47         $data['mysql_time'] = $this->aTIMES['MySQL']['sum'];
48         $data['mysql_count_queries'] = $this->aTIMES['MySQL']['cnt'];
49         $data['mysql_queries'] = $this->aTIMES['MySQL']['comment'];
50         $data['sphinx_time'] = $this->aTIMES['Sphinx']['sum'];
52         $query_logger->logProfilingData($data);
54     }
56     // This helper function implements the Singleton pattern
57     function getInstance( ) {
58         static $instance;
59
60         if(!isset($instance)) {
61             $instance = new Timer( );
62         }
64         return($instance);
65     }
66 }
67 ?>

```

在应用中使用 Timer class 是很方便的，只要使用一个 Timer 包裹开销巨大或让某些用户感兴趣的调用即可。例如，下面演示了如何在每个 MySQL 查询上包裹一个 Timer。

PHP 新的 mysqli 接口允许用户扩展基本的 mysqli 类，并且重新声明查询方法。

```

68 <?php
69 class mysqlx extends mysqli {
70     function query($query, $resultmode) {
71         $timer = Timer::getInstance( );
72         $timer->startTime('MySQL');
73         $res = parent::query($query, $resultmode);
74         $timer->stopTime('MySQL', "Query: $query\n");
75         return $res;
76     }
77 }
78 ?>

```

这种技术手段只需要改变很少的代码，就可以全局性地，简单地将 mysqli 转换为 mysqlx，而整个应用记录所

有的查询。使用这种方法，可以测量所有针对任何外部资源的访问，例如那些对 Sphinx 全文搜索引擎的查询。

```
$timer->startTime('Sphinx');
$this->sphinxres = $this->sphinx_client->Query ( $query, "index" );
$timer->stopTime('Sphinx', "Query: $query\n");
```

接着，看一下如何记录搜集来的数据，下面有一个例子演示了何时使用 MyISAM 存储引擎，或者归档存储引擎。这两个引擎都适用于存储日志记录。在为日志增加一条记录时，使用了 INSERT DELAYED 语句，这种 INSERT 语句，会在数据库服务器做为一个后台线程被执行。这意味着被测试的查询将直接返回查询结果，而不会对应用的响应时间造成明显影响。（即便不使用 INSERT DELAYED，插入操作也会并发地进行，除非显式地屏蔽它，因此来自外部的 SELECT 查询不会阻塞日志的记录过程）。最后，顺便创建一个基于每天日期的新日志表（Log Table），实现一种基于日期的分区方案（Scheme）。

用 CREATE TABLE 语句创建日志记录表。

```
CREATE TABLE logs.performance_log_template (
  ip                INT UNSIGNED NOT NULL,
  page              VARCHAR(255) NOT NULL,
  utime             FLOAT NOT NULL,
  wtime             FLOAT NOT NULL,
  mysql_time        FLOAT NOT NULL,
  sphinx_time       FLOAT NOT NULL,
  mysql_count_queries INT UNSIGNED NOT NULL,
  mysql_queries     TEXT NOT NULL,
  stime             FLOAT NOT NULL,
  logged            TIMESTAMP NOT NULL
                    default CURRENT_TIMESTAMP on update CURRENT_TIMESTAMP,
  user_agent        VARCHAR(255) NOT NULL,
  referer           VARCHAR(255) NOT NULL
) ENGINE=ARCHIVE;
```

实际上在这张表中不会插入任何数据，这只是一个模板，这里将使用 CREATE TABLE LIKE 语句，依据此模板为每天的数据创建一个表。

在第 3 章时还将深入讨论此问题，但在这里，只需要记住使用最小数据类型存放数据即可。这里使用无符号整型来存放 IP 地址，使用 255 个字符长度的字段存放页面和相关引用。实际的值可以大于 255 个字符长度，但通常有前 255 个字符就足以满足需求了。

代码最后一部分是在页面执行完毕记录相关结果，这里有相关 PHP 代码用来记录数据。

```
79 <?php
80 // Start of the page execution
81 $timer = Timer::getInstance( );
82 $timer->startTime('Page');
83 // ... other code ...
84 // End of the page execution
85 $timer->stopTime('Page');
86 $timer->logdata( );
87 ?>
```

计时类（Timer class）引用了 DBQueryLog helper 类，它负责记录数据到数据库，创建每天的新日志表。其代码如下。

```
88 <?php
89 /*
90  * Class DBQueryLog logs profiling data into the database
91  */
```



```

92 class DBQueryLog {
93
94     // constructor, etc, etc...
95
96     /*
97     * Logs the data, creating the log table if it doesn't exist. Note
98     * that it's cheaper to assume the table exists, and catch the error
99     * if it doesn't, than to check for its existence with every query.
100    */
101    function logProfilingData($data) {
102        $table_name = "logs.performance_log_" . @date("ymd");
103
104        $query = "INSERT DELAYED INTO $table_name (ip, page, utime,
105            wtime, stime, mysql_time, sphinx_time, mysql_count_queries,
106            mysql_queries, user_agent, referer) VALUES (.. data ..)";
107        $res = $this->mysqlx->query($query);
108        // Handle "table not found" error - create new table for each new day
109        if ((!$res) && ($this->mysqlx->errno == 1146)) { // 1146 is table not found
110            $res = $this->mysqlx->query(
111                "CREATE TABLE $table_name LIKE logs.performance_log_template");
112            $res = $this->mysqlx->query($query);
113        }
114    }
115 }
116 }
117 ?>

```

一旦记录了一定量的数据，就可以进行日志分析了。使用 MySQL 记录日志的妙处是可以使用灵活的 SQL 语句来对其进行分析，可以很容易地写一些查询来访问日志记录信息，生成任何所需要的报告。例如，找出 2007 年 2 月的第一天，执行时间超过 10 秒钟的页面记录：

```

mysql> SELECT page, wtime, mysql_time
-> FROM performance_log_070201 WHERE wtime > 10 LIMIT 7;
+-----+-----+-----+
| page                                | wtime  | mysql_time |
+-----+-----+-----+
| /page1.php                          | 50.9295 | 0.000309   |
| /page1.php                          | 32.0893 | 0.000305   |
| /page1.php                          | 40.4209 | 0.000302   |
| /page3.php                          | 11.5834 | 0.000306   |
| /login.php                          | 28.5507 | 28.5257    |
| /access.php                         | 13.0308 | 13.0064    |
| /page4.php                          | 32.0687 | 0.000333   |
+-----+-----+-----+

```

（在这个查询里，实际上还有更多数据未列出，这里只是为了演示故意裁短了。）

如果想比较 wtime（挂钟时间）和查询时间的差别，可以发现 MySQL 的查询时间变慢的原因，是 7 个页面中的两个页面的响应时间比较慢。性能分析数据中已经存储了相关查询记录，可以在检查过程中，专门提取这些查询信息。

```

mysql> SELECT mysql_queries
-> FROM performance_log_070201 WHERE mysql_time > 10 LIMIT 1\G
***** 1. row *****
mysql_queries:
Query: SELECT id, chunk_id FROM domain WHERE domain = 'domain.com'
Time: 0.00022602081298828
Query: SELECT server.id sid, ip, user, password, domain_map.id as chunk_id FROM
server JOIN domain_map ON (server.id = domain_map.master_id) WHERE domain_map.id = 24
Time: 0.00020599365234375
Query: SELECT id, chunk_id, base_url,title FROM site WHERE id = 13832

```

```

Time: 0.00017690658569336
Query: SELECT server.id sid, ip, user, password, site_map.id as chunk_id FROM server
JOIN site_map ON (server.id = site_map.master_id) WHERE site_map.id = 64
Time: 0.0001990795135498
Query: SELECT from_site_id, url_from, count(*) cnt FROM link24.link_in24 FORCE INDEX
(domain_message) WHERE domain_id=435377 AND message_day IN (...) GROUP BY from_site_
id ORDER BY cnt desc LIMIT 10
Time: 6.3193740844727
Query: SELECT revert_domain, domain_id, count(*) cnt FROM art64.link_out64 WHERE
from_site_id=13832 AND message_day IN (...) GROUP BY domain_id ORDER BY cnt desc
LIMIT 10
Time: 21.3649559021

```

上面结果中发现了两个问题查询，一个执行时间为 6.3 秒，另一个为 21.3 秒，这两个查询需要优化。

2

在上述方式中记录所有查询是开销很大的，所以通常只记录页面记录的一小部分，或者是使用调试 (Debug) 模式。

如何判断性能瓶颈是否来自未进行性能分析的系统的某个部分？有一个很简单的方法——检查“丢失时间 (Lost Time)”。一般来说，挂钟时间 (Wtime) 是用户时间 (User Time)、系统时间 (System Time)、SQL 查询时间，其他可测量的时间，加上“丢失时间”的总和，“丢失时间”是无法测量的。当然，这里时间分类有些重叠，比如 PHP 代码处理 SQL 查询占用的 CPU 时间，但这通常是可忽略的。表 2-2 是一个理想化的演示，表示如何分割挂钟时间的各个组成。

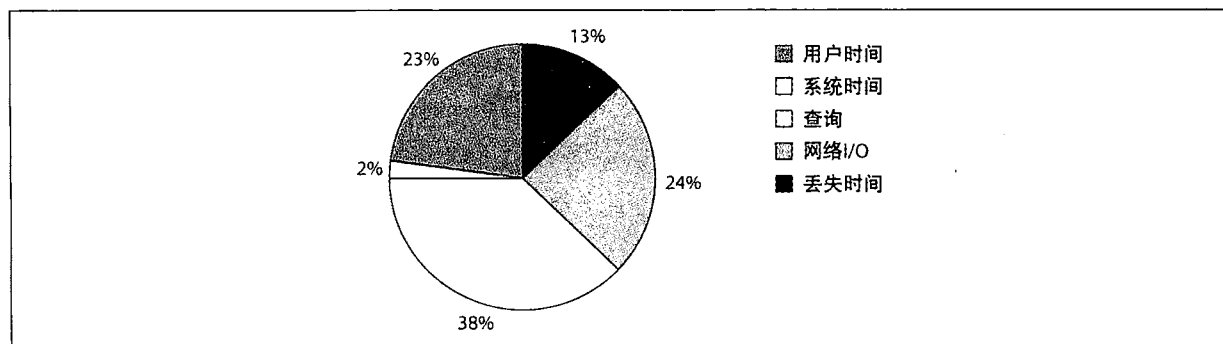


图 2-2: 丢失时间是记录时间和墙钟时间 (Wall-clock Time) 之间的差值

理想的状态下，“丢失时间”越小越好。如果挂钟时间减去所有被测量的时间，还有很大剩余，那么这种剩余可能是脚本执行增加的时间，也可能是产生页面增加的时间，或者是某个步骤产生的时延 (注 6)。

有两种类型等待：在队列中等待获得 CPU 处理时间，或者等待某些资源。一个进程运行前会在队列中等待，但可能所有 CPU 都在忙。通常来说，不太可能计算出一个进程在 CPU 等待队列要花多长时间，但这不是个问题。更多的情况是可能在处理外部资源调用，而无法进行性能分析。

如果性能分析是非常完整的，应该很容易发现性能瓶颈所在。一般来说这非常直观：如果脚本执行时间大多是 CPU 占用时间，那么可能需要检查和优化 PHP 代码。不过有时某些测量值也会影响其他测量值。

例如，表面看 CPU 占用率非常高，但存在某种 bug，如缓存 (Cache) 系统效率低下，而导致应用必须做更多的 SQL 查询，耗费了很多 CPU 时间。

3

注 6: 假定 Web 服务器能缓冲结果，这样脚本就会自动结束，发送到客户端的所消耗的时间也不会被计算。

如上述例子演示的，在应用级别进行性能分析是种非常灵活而有用的技术。如果可能，为了定位性能瓶颈，在必要的地方为应用插入性能分析代码，是个很好的解决方法。

最后说明一下，以上演示的只是基本的应用性能分析技术，而本节的目标是帮助判定 MySQL 是否存在性能问题，可能还需要对应用代码进行深入的性能分析。例如，如果应用占用 CPU 时间太多，可能需要优化 PHP 代码。可以使用 xdebug、Valgrind 和 cachegrind 等工具，对 CPU 用的占用率进行性能分析。

有些语言对性能分析有内嵌的支持功能，例如，可以在 Ruby 代码中，使用 -r 命令行加入性能分析，或者在 Perl 代码中使用下列方法加入性能分析。

```
$ perl -d:DProf <script file>
$ dprofpp tmon.out
```

在网络上搜索“profiling <某种语言>”是个不错的了解开始。

2.5.2 MySQL 分析

MySQL Profiling

我们将会详细讨论 MySQL 分析，因为它和特定应用程序关系不大。应用程序分析和服务器分析有时都是需要的。尽管应用程序分析可以更完整地展示整个系统的性能，但分析 MySQL 能提供在对应用程序进行整体分析时无法得到的大量信息。例如，分析 PHP 代码不会显示查询将会检查多少行代码。

和应用程序分析一样，分析 MySQL 的目的在于找到耗费了最多时间的工作。我们不会分析 MySQL 的源代码，尽管那对 MySQL 定制安装会有好处，但那是另外一本书的内容。相反地，我们展示了一些可以用来捕获和分析 MySQL 为了执行查询所进行的不同工作的信息。

你可以工作在任何适合你需要的粒度上：可以整体分析服务器，或者检查单个查询或批查询。可以得到的信息包括：

- MySQL 访问得最多的数据。
- MySQL 执行得最多的查询的种类。
- MySQL 停留时间最长的状态。
- MySQL 用来执行查询的使用得最频繁的子系统。
- MySQL 查询过程中访问的数据种类。
- MySQL 执行了多少种不同类型的活动，比如索引扫描。

我们从分析整个服务器这一最宽泛的层面出发，逐步朝细节迈进。

记录查询

MySQL 有两种查询日志：普通日志（General Log）和慢速日志（Slow Log）。它们都会记录查询，但是位于查询执行过程相对的两头。普通日志记录了服务器接收到的每一个查询，因此它包含了没有被执行的查询，比如因为错误而未被执行的查询。普通日志捕获了所有查询，还有一些非查询事件，比如连接和断开连接。可以使用一个配置命令打开它：

```
log = <file_name>
```

根据设计，普通日志不包含执行时间或其他只有在查询结束之后才能得到的信息。相反地，慢速日志只包含了已经执行过的查询。详细说来，它记录了执行时间超过了特定长度的查询。两种日志都有助于分析，但是慢速日志是找到问题查询的主要工具。我们通常推荐打开它。

下面的配置示例打开了日志，捕获了所有执行时间超过 2 秒的查询，并且记录了没有使用索引的查询。它也将记录执行速度较慢的管理命令，比如 `OPTIMIZE TABLE`：

```
log-slow-queries          = <file_name>
long_query_time           = 2
log-queries-not-using-indexes
log-slow-admin-statements
```

你可以修改这个示例，然后把它放在 `my.cnf` 配置文件中。更多关于服务器配置的知识请参阅第 6 章。

`Long_query_time` 的默认时间是 10 秒。这对于大多数配置来说都太长了，通常使用 2 秒。然而，即使是 1 秒对于许多场景来说也太长了。下节我们会介绍对它进行精密调整的方法。

在 MySQL5.1 中，全局性的 `slow_query_log` 和 `slow_query_log_file` 这两个系统变量提供了对于慢速查询日志的运行控制，但是在 MySQL5.0 中，不能在未重启 MySQL 服务器的情况下打开或关闭慢速查询日志。对于 MySQL5.0，通常的临时解决办法是使用 `long_query_time` 变量，它是可以动态设置的。下面的命令不会真正地关闭慢速日志记录，但是它实际上有同样的效果（如果任何查询的执行时间超过了 10 000 秒的，那肯定意味着需要优化！）：

```
mysql> SET GLOBAL long_query_time = 10000;
```

一个相关的配置变量是 `log_queries_not_using_indexes`，它使服务器把没有使用索引的查询记录到慢速查询日志中，无论它们执行速度有多快。尽管打开慢速日志相对于执行慢速查询来说，通常只增加了一点点时间，但是没有使用索引的查询都非常快（例如扫描很小的表），因此记录它们会导致服务器变慢，甚至还会使用大量的磁盘空间。

不幸的是，你不能在 MySQL5.0 中使用动态的可设置变量对某个查询打开或关闭记录。你必须编辑配置文件，然后重启 MySQL。一种减少负担，但却不用重启服务器的办法是在希望关闭日志的时候让日志文件成为 `/dev/null` 的符号链接（Symbolic Link）（实际上，可以对任何日志文件都使用这种技巧）。你只需要在改变后运行 `FLUSH LOGS` 以确保 MySQL 关闭当前的日志文件描述符，并且重新把文件打开到 `/dev/null`。

和 MySQL5.0 不同，MySQL5.1 可以在运行时改变日志行为并且记录到数据表中，然后用 SQL 语句进行查询。这是一个很大的改进。

更精细地控制日志

MySQL5.0 和之前版本中的慢速查询日志有一些缺陷，使它不适用于某些场合。一个问题就是它的粒度只能到秒，并且 MySQL5.0 中 `long_query_time` 的最小值为 1 秒。对于大多数交互性的应用程序来说，这个值太长了。如果开发高性能的网页应用程序，那么你会期望生成整个页面的时间远小于 1 秒，并且页面在生成的时候还会发起很多查询。在这种情况下，一个执行时间为 150 毫秒的查询也许就够慢的了。

另外一个问题就是不能把服务器执行的所有查询记录到慢速日志中（特别是从服务器线程的查询不会被记录）。普通日志记录了所有的查询，但是它在解析查询之前就记录了下来，因此它不会包含诸如执行时间、锁定时间、检查行数等信息。只有慢速查询日志才有这些信息。

最后，如果开启了 `log_queries_not_using_indexes` 选项，慢速日志也许就会被那些快速高效的全表扫描查询塞满。例如，如果使用 `SELECT * FROM STATES` 产生了一个关于 `states` 的下拉列表，由于查询执行了全表扫描，它会被记录下来。

出于性能优化的目的进行分析，其实就是在寻找导致服务器最大工作量的查询。这并不总是意味着慢速查询，因此记录“慢速”查询的想法并不是很有用。比如，每秒运行 1000 个耗时 10 毫秒的查询给服务器造成的负担会高于每秒运行一个耗时 10 秒的查询。为了确定这样的问题，就需要记录每个查询并且分析结果。

既研究造成服务器最多工作的查询，又研究慢速查询（即使它们并不经常被执行）通常是个好主意。这有助于发现不同类型的问题，例如导致不好用户体验的查询。

99 在 Georg Richter 工作的基础上，我们为 MySQL 服务器开发了一个补丁，它可以用毫秒精度定义慢速查询日志，而不是秒。它还可通过设置 `long_query_time=0`，把所有查询记录到慢速日志中。可以到 <http://www.mysqlperformanceblog.com/mysql-patches/> 下载该补丁。它主要的缺点就是为了使用它，你不得不自己编译 MySQL，因为补丁在 MySQL5.1 之前的版本中没有被包含在官方的发布中。

在写作本书的时候，MySQL5.1 中包含的版本仅仅改变了时间粒度。该补丁的新版本没有被包含在任何 MySQL 官方发布中，但增添了好些有用的新功能。它包含了查询的连接 ID，还有关于查询缓存、联接类型、临时表和排序方面的信息。它也增加了 InnoDB 统计，例如 I/O 活动方面的信息和锁等待的信息。

新补丁使你可以记录由从 SQL 线程执行的查询，它对于排查从服务器复制（Replication）滞后问题非常重要。

（关于如何让从服务器跟上主服务器的更多内容请参阅第 399 页“过大的复制延迟”）。它还支持有选择性地记录某些会话。这对于分析目的来说已经足够，对于实际情况也很有好处。

这个补丁相对较新，因此在使用的时候要小心一些。我们认为它很安全，但是它没有像 MySQL 服务器其他部分那样经历过很多考验。如果担心它的稳定性，那么就不用在所有时候都运行这个补丁，只需要在记录某些查询的时候启动补丁就可以了，其他时候可以把它关掉。

在分析的时候，使用 `long_query_time=0` 来记录所有查询是有用的。如果大部分负载都来自简单的查询，那么就能知道原因。记录所有的查询会对性能造成一点影响，并且还需要很多磁盘空间，这是不希望所有时候都记录所有查询的另外一个原因。幸运的是，可以在不重启服务器的情况下改变 `long_query_time` 的值，这样就能得到短时间内所有查询的样本，然后复原日志，只记录很慢的查询。

如何阅读慢速查询日志

下面有一个慢速查询日志的样本：

```
1 # Time: 030303 0:51:27
2 # User@Host: root[root] @ localhost []
3 # Query_time: 25 Lock_time: 0 Rows_sent: 3949 Rows_examined: 378036
4 SELECT ...
```

第 1 行表示查询被记录的时间，第 2 行显示了执行查询的用户。第 3 行显示了执行的时间、在 MySQL 服务器阶段（不是在存储引擎阶段）等待表锁的时间、查询返回的行数，以及查询检查的行数。

99 这些行都被注释掉了，因此当把它们拷贝到 MySQL 客户端的时候，它们不会被执行。最后一行是查询。

下面是 MySQL5.1 服务器的例子：

```

1 # Time: 070518 9:47:00
2 # User@Host: root[root] @ localhost []
3 # Query_time: 0.000652 Lock_time: 0.000109 Rows_sent: 1 Rows_examined: 1
4 SELECT ...

```

除了第 3 行的时间精度之外，大部分信息都是一样的。较新的补丁添加了更多信息：

```

1 # Time: 071031 20:03:16
2 # User@Host: root[root] @ localhost []
3 # Thread_id: 4
4 # Query_time: 0.503016 Lock_time: 0.000048 Rows_sent: 56 Rows_examined: 1113
5 # QC_Hit: No Full_scan: No Full_join: No Tmp_table: Yes Disk_tmp_table: No
6 # Filesort: Yes Disk_filesort: No Merge_passes: 0
7 # InnoDB_IO_r_ops: 19 InnoDB_IO_r_bytes: 311296 InnoDB_IO_r_wait: 0.382176
8 # InnoDB_rec_lock_wait: 0.000000 InnoDB_queue_wait: 0.067538
9 # InnoDB_pages_distinct: 20
10 SELECT ...

```

第 5 行显示了查询是否从查询缓存中取值，是否进行了全表扫描，是否进行了没使用索引的联接，是否使用了临时表，如果是，那么是否使用了磁盘上的临时表。第 6 行显示了查询是否使用了文件排序，如果是，那么它是否在磁盘上进行了排序，排序的时候执行了多少次排序合并。

第 7、8、9 行在使用 InnoDB 的时候会出现。第 7 行显示了在查询期间 InnoDB 安排了多少次页面读取操作，以及相应的以字节表示的值。第 7 行的最后一个值是 InnoDB 从磁盘上读取数据花费的时间。第 8 行显示了查询等待行级锁的时间和等待进入 InnoDB 核心 (InnoDB Kernel) 的时间 (注 7)。

第 9 行显示了查询大致访问了多少个唯一的 InnoDB 页面。这个数字增长得越大，精确性就越低。这个数字的一个用途是估计查询工作集的数据页，它说明了 InnoDB 缓冲池如何缓存数据。它也显示了聚集索引的巨大帮助。如果查询的行很好地被聚集到了一起，访问的页面就会较少。更多内容请参看第 110 页“聚集索引”。

使用慢速查询日志来诊断慢速查询并不总是那么显而易见。尽管日志包含了大量的有用信息，但是遗漏了一个重要的信息：查询为什么会慢。有时道理很明显。如果日志显示检查了 12 000 000 行数据，并且向客户发送了 1 200 000 行数据，那么慢的原因也就很清楚了，因为这是一个巨大的查询。然而，很少有如此直白的时候。要注意不要过于注重慢速查询日志的含义。如果发现同样的查询在日志中出现了很多次，那么就说明了它很慢，需要优化。但是查询在日志中只出现了一次并不意味着它是一个不好的查询。你也许会发现一个慢速查询，然后自己运行一下，发现运行时间才几分之一秒。日志中出现查询说明它当时运行得慢，但是未必意味着现在就慢，也不意味着将来也会慢。查询时快时慢有很多原因：

88

- 表也许被锁定了，导致查询处于等待状态。Lock_time 显示了查询等待锁被释放的时间。
- 数据或索引也许没有被缓存在内存中。对于第一次启动服务器或服务器没有调优，这是常见情况。
- 晚间备份正在进行中，使所有磁盘 I/O 变慢。
- 服务器也许同时在运行其他的查询，减慢了当前查询。

这样说来，只能把慢速查询日志看成调查工作的一部分。可以用它来找到很多可疑的查询，但是要要对它们进行仔细地排查。

慢速查询日志补丁的特定目的是帮助你理解查询为何缓慢，尤其是使用 InnoDB 的时候，它的统计有很大的帮助。从 InnoDB 的统计中，你可以知道查询是否正在等待磁盘 I/O，是否在 InnoDB 队列中等待了大量的时间，等等。

注 7：更多关于 InnoDB 核心的内容请参阅第 296 页“InnoDB 并发调优”。

日志分析工具

一旦记录了某些查询，那么接下来就要分析它们。通常的策略是找到对服务器影响最大的查询，使用 `EXPLAIN` 检查它们的执行计划，并且按照需要进行调优。在调优之后要重复分析过程，因为改动可能会影响别的查询。例如，索引通常对 `SELECT` 查询有帮助，但是会减缓 `INSERT` 和 `UPDATE` 查询。

通常应该在日志中查找下面 3 个信息：

长查询

日常的批处理工作将会产生很长的查询，但普通的查询不会很长。

影响很大的查询

寻找消耗了服务器大部分执行时间的查询，取消那些通常需要大量执行时间的短小查询。

新查询

查找今天的查询，而不是过去 100 天中最慢的查询。它们有可能是新查询，也有可能是以前运行得很快查询，但是由于索引或其他变化，现在变慢了。

如果慢速查询日志很小，就很容易进行手工分析，但是如果记录了所有查询（正如我们建议的那样），这时就需要工具了。下面有一些分析日志的常用工具：

Mysqldumpslow

MySQL 在服务器中提供了 `mysqldumpslow`。它是一个 Perl 脚本，可以总结慢速查询日志，并且显示每个查询在日志中出现的次数。这样，就不用浪费时间去分析一个一天只出现了一次的耗时 30 秒的查询，而去分析那些每天都运行了上千次的查询。

`Mysqldumpslow` 的好处是它已经安装到机器上了，它的缺点是灵活性比其他工具差。它也没什么文档，并且不能理解毫秒精度的慢速查询日志。

Mysql_slow_log_filter

可以从 http://www.mysqlperformanceblog.com/files/utils/mysql_slow_log_filter 下载到该工具，它能理解毫秒精度的日志。可以用它来提取执行时间超过某个阈值的查询，或者检查数据行数超过特定数目的查询。它特别适合于对使用了毫秒补丁的日志进行“追踪”，因为该补丁会让日志增长得非常快，如果不进行过滤，根本无法进行分析。可以给该工具设定一个较高的阈值运行一会儿，直到所有的冲突因素都消失掉，然后再改变参数捕捉查询并且继续调优。

下面的命令会显示运行时间长于 0.5 秒或检查行数大于 1 000 的查询：

```
$ tail -f mysql-slow.log | mysql_slow_log_filter -T 0.5 -R 1000
```

Mysql_slow_log_parser

这是另外一个工具，可以从 http://www.mysqlperformanceblog.com/files/utils/mysql_slow_log_parser 下载，它能对毫秒精度的慢速日志进行聚合。除了聚合和报表之外，它还能显示执行时间和分析行数的最大和最小值，打印出“规范化”的查询，并且打印出可以使用 `EXPLAIN` 的进行分析的示例。下面是它输出的例子：


```

### 3579 Queries
### Total time: 3.348823, Average time: 0.000935686784017883
### Taking 0.000269 to 0.130820 seconds to complete
### Rows analyzed 1 - 1
SELECT id FROM forum WHERE id=XXX;
SELECT id FROM forum WHERE id=12345;

```

Mysqslsla

它的名称是 MySQL 命令日志分析工具 (MySQL Statement Log Analyzer), 可以从 <http://hackmysql.com/mysqslsla> 下载。它可以分析慢速日志, 还可以分析普通日志和包含分隔的 SQL 命令的“原始”日志。和 mysql_slow_log_parser 一样, 它可以进行规范化和汇总。它也能利用 EXPLAIN 分析查询 (它针对 EXPLAIN 重写了许多非 SELECT 命令) 并产生复杂的报告。

可以使用慢速日志统计来预测服务器资源消耗减少的数量。假设在一个小时之内 (3 600 秒) 对查询进行抽样, 并发现日志中所有查询消耗时间的总和是 10 000 秒 (总时间大于挂钟时间, 因为查询是并行执行的)。如果日志分析显示最差的查询消耗的执行时间为 3 000 秒, 那么就可以知道该查询占总负载的 30%, 由此得知通过优化查询可以为服务器节约多少资源。

2.5.3 分析 MySQL 服务器

为了了解服务器在什么任务上花费了最多时间, 一个最好的分析方法是使用 SHOW STATUS。SHOW STATUS 返回了大量状态信息, 我们在这儿只说明它输出的一些变量。



提示: SHOW STATUS 有一些具有欺骗性的行为, 会在 MySQL5.0 和更新的版本中给出错误的结果。更多关于 SHOW STATUS 的缺点和行为请参阅第 13 章。

为了近似于实时地知道服务器性能, 可以周期性地运行 SHOW STATUS, 并且和前一次的输出进行比较。可以用下面的命令进行这项工作:

```
mysqladmin extended -r -i 10
```

一些变量不会严格地增加计数器, 因此会看到一些古怪的输出, 比如 Threads_running 为负数。这没什么关系, 它只是说明该项结果相比上次是下降的。

因为输出很多, 所以可以把结果导入到 grep 中, 过滤掉自己不想看的变量。也可以使用 innotop 或第 14 章中提到的其他工具来检查结果。一些值得监控的更有用的变量是:

Bytes_received 和 Bytes_sent

和服务器之间来往的流量。

Com_*

服务器正在执行的命令。

Created_*

在查询执行期间创建的临时表和文件。

71 Handler_*
 存储引擎操作。

Select_*
 不同类型的联接执行计划。

Sort_*
 几种排序信息。

可以用这种方法监视 MySQL 的内部操作，例如键访问的次数、为 MyISAM 从磁盘上进行的键读取、数据访问率、为 InnoDB 从磁盘上读取的数据，等等。这有助于定位系统中实际的和潜在的瓶颈，而无需调查每一个查询。也可以用 `mysqlreport` 这样的工具来分析 `SHOW STATUS` 产生的结果，得到服务器总体健康状况的快照。

我们这儿不会详细讨论各个状态变量的含义，但是会在涉及它们的例子里面进行解释，因此不用担心自己不理解它们的含义。

分析 MySQL 服务器的另外一种好办法是使用 `SHOW PROCESSLIST`。它不仅可以显示哪种查询正在执行，也能看到连接的状态。一些因素，比如大量连接处于锁定状态，是瓶颈的明显线索。和 `SHOW STATUS` 一样，`SHOW PROCESSLIST` 的输出非常详尽，以至于使用 `innotop` 这样的工具进行分析会比手工分析方便得多。

2.5.4 使用 SHOW STATUS 分析查询

`FLUSH STATUS` 和 `SHOW SESSION STATUS` 相结合对分析 MySQL 执行的查询或批处理查询非常有帮助。它是优化查询的好办法。

先看看一个例子。首先，运行 `FLUSH STATUS` 把会话状态变量设置为零，这样就可以知道 MySQL 执行查询时做了多少工作：

```
mysql> FLUSH STATUS;
```

接下来运行查询。我们添上了 `SQL_NO_CACHE`，这样 MySQL 不会从查询缓存中取得查询结果：

```
mysql> SELECT SQL_NO_CACHE film_actor.actor_id, COUNT(*)
-> FROM sakila.film_actor
-> INNER JOIN sakila.actor USING(actor_id)
-> GROUP BY film_actor.actor_id
-> ORDER BY COUNT(*) DESC;
...
200 rows in set (0.18 sec)
```

查询返回了 200 行结果，但是它到底做了什么？`SHOW STATUS` 能显示详细状态。先看看服务器选择的查询计划：

```
mysql> SHOW SESSION STATUS LIKE 'Select%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Select_full_join | 0 |
| Select_full_range_join | 0 |
| Select_range | 0 |
| Select_range_check | 0 |
| Select_scan | 2 |
+-----+-----+
```

看上去 MySQL 进行了全表扫描（实际上，看上去进行了两次扫描，但是这是 SHOW STATUS 造成的错觉，我们稍后再说）。如果查询涉及了多张表，有几个变量的值就会大于零。例如，如果 MySQL 在后续表中进行了范围扫描，以寻找匹配行，Select_full_range_join 就会有值。甚至还可以查看查询执行的低层次存储引擎操作：

```
mysql> SHOW SESSION STATUS LIKE 'Handler%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Handler_commit | 0     |
| Handler_delete | 0     |
| Handler_discover | 0     |
| Handler_prepare | 0     |
| Handler_read_first | 1     |
| Handler_read_key | 5665  |
| Handler_read_next | 5662  |
| Handler_read_prev | 0     |
| Handler_read_rnd | 200   |
| Handler_read_rnd_next | 207   |
| Handler_rollback | 0     |
| Handler_savepoint | 0     |
| Handler_savepoint_rollback | 0     |
| Handler_update | 5262  |
| Handler_write | 219   |
+-----+-----+
```

“读取 (Read)” 操作的值很高，意味着 MySQL 需要扫描多个表才能满足查询需要。通常，如果 MySQL 只对一个表使用了全表扫描，我们就会看到 Handler_read_rn_next 的值较高，并且 Handler_read_rnd 是零。

在这个例子中，多个非零值意味着 MySQL 必须使用临时表来满足 GROUP BY 和 ORDER BY 子句。这是 Handler_write 和 Handler_update 不为零的原因：MySQL 假定写入临时表，扫描它并进行排序，然后再次进行扫描，输出排序后的结果。再来看看 MySQL 为排序做了些什么：

```
mysql> SHOW SESSION STATUS LIKE 'Sort%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Sort_merge_passes | 0     |
| Sort_range | 0     |
| Sort_rows | 200   |
| Sort_scan | 1     |
+-----+-----+
```

正如我们猜测的那样，MySQL 通过扫描包含输出中所有行的临时表进行排序。如果值多于 200 行，我们怀疑它在查询执行的过程中在别的地方进行了排序。还能看到 MySQL 为查询创建了多少临时表：

```
mysql> SHOW SESSION STATUS LIKE 'Created%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Created_tmp_disk_tables | 0     |
| Created_tmp_files | 0     |
| Created_tmp_tables | 5     |
+-----+-----+
```

很高兴看到查询不需要使用磁盘上的临时表，因为它很慢。但是还是有一点小小的疑问，确定 MySQL 没有为这个查询创建 5 个临时表吗？

事实上，这个查询值需要一个临时表。这也是我们之前提到过的假信息。发生了什么事情？这个例子运行在

MySQL5.0.45 上, 在 MySQL5.0 版本中, SHOW STATUS 实际上从 INFORMATION_SCHEMA 表选择数据, 它引入了所谓的“观察的代价”(注 8)。如果再运行一次 SHOW STATUS, 结果还会有小小的不一样:

```
mysql> SHOW SESSION STATUS LIKE 'Created%';
+-----+-----+
| Variable_name          | Value |
+-----+-----+
| Created_tmp_disk_tables | 0     |
| Created_tmp_files      | 0     |
| Created_tmp_tables     | 6     |
+-----+-----+
```

注意那个值再次增加了, Handler 和其他变量也一样被影响到了。根据 MySQL 版本不同, 结果也会不同。

也可以在 MySQL4.1 和之前的版本中采用同样的流程, 先运行 FLUSH STATUS, 运行查询并运行 SHOW STATUS。这需要一个空闲的服务器, 因为老版本有全局变量, 它会被其他进程改变。

弥补由于运行 SHOW STATUS 带来的“观察的代价”的最佳方式是将命令运行两次, 并从第 2 次的结果中减去第 1 次的结果来计算开销。为了得到精确的结果, 你需要知道变量的范围, 这样就可以知道观察的代价, 一些是针对每次会话的, 另外一些是全局性的。可以用 mk-query-profiler 自动化这个复杂的过程。

可以把这种自动化的分析过程整合到应用程序的数据库联接代码中。当机会分析的时候, 联接代码在每个查询执行之前会自动清除状态并且在执行之后记录区别。另外一种替代的方式是分析每个页面, 而不是每个查询。无论哪种策略都有助于显示 MySQL 在执行查询的过程中做了多少工作。

2.5.5 SHOW PROFILE

SHOW PROFILE 是一个由 Jeremy Cole 贡献给社区的补丁, 主要针对 5.0.37 版(注 9)。在默认情况下, 分析是被关闭的, 但是可以在会话的层面打开。打开它会让服务器收集用于执行查询的资源的信息。在开始收集统计信息之前, 需要把分析变量设置为 1:

```
mysql> SET profiling = 1;
```

现在运行查询:

```
mysql> SELECT COUNT(DISTINCT actor.first_name) AS cnt_name, COUNT(*) AS cnt
-> FROM sakila.film_actor
-> INNER JOIN sakila.actor USING(actor_id)
-> GROUP BY sakila.film_actor.film_id
-> ORDER BY cnt_name DESC;
...
997 rows in set (0.03 sec)
```

查询的分析数据被保存在会话中。使用 SHOW PROFILES 查看已经被分析过的查询:

```
mysql> SHOW PROFILES\G
***** 1. row *****
Query_ID: 1
Duration: 0.02596900
Query: SELECT COUNT(DISTINCT actor.first_name) AS cnt_name,...
```

注 8: “观察的代价”问题在 MySQL5.1 中的 SHOW SESSION STATUS 命令中得到了修复。

注 9: 在写作本书的时候, 尽管 MySQL 企业版的版本高于 5.0.37, 这个补丁也没有被官方收录。

可以使用 SHOW PROFILE 命令取得被保存下来的分析数据。如果不加任何参数，它就会显示最近一个命令的状态值和运行时间：

```
mysql> SHOW PROFILE;
+-----+-----+
| Status                               | Duration |
+-----+-----+
| (initialization)                     | 0.000005 |
| Opening tables                        | 0.000033 |
| System lock                           | 0.000037 |
| Table lock                            | 0.000024 |
| init                                  | 0.000079 |
| optimizing                            | 0.000024 |
| statistics                            | 0.000079 |
| preparing                             | 0.000003 |
| Creating tmp table                    | 0.000124 |
| executing                             | 0.000008 |
| Copying to tmp table                  | 0.010048 |
| Creating sort index                   | 0.004769 |
| Copying to group table                | 0.0084880 |
| Sorting result                        | 0.001136 |
| Sending data                          | 0.000925 |
| end                                   | 0.000001 |
| removing tmp table                    | 0.000004 |
| end                                   | 0.000005 |
| removing tmp table                    | 0.000001 |
| end                                   | 0.000011 |
| query end                             | 0.000001 |
| freeing items                         | 0.000025 |
| removing tmp table                    | 0.000001 |
| freeing items                         | 0.000016 |
| closing tables                       | 0.000017 |
| logging slow query                    | 0.000006 |
+-----+-----+
```

75

每一行代表了进程的一种变化,以及在这种状态停留的时间。Status 列和 SHOW FULL PROCESSLIST 输出的 State 列是对应的。它的值来自于 thd->proc_info 变量，因此可以直接看到 MySQL 内部的值。尽管它们的名字都很直观，也不难理解，但是这些变量还是可以在 MySQL 的手册中找到。

可以从 SHOW PROFILES 的输出中得到特定的 Query_ID，并且进行指定的分析，并且还可以定义输出的其余列。例如，为了了解执行查询时用户的 CPU 使用率，可以使用下面的命令：

```
mysql> SHOW PROFILE CPU FOR QUERY 1;
```

SHOW PROFILE 很好地揭示了服务器执行查询时所做的事情，并且有助于理解查询在操作上花费时间。它的局限是未实现的特性，不能查看和分析其他联接的查询，以及由于分析带来的开销。

2.5.6 使用另外的手段分析 MySQL



本章我们已经向你展示了如何使用 MySQL 的内部状态信息来显示服务器内部所做的事情，你也可以使用 MySQL 另外的状态输出来进行一些分析。另外还有一些有用的命令，包括 SHOW INNODB STATUS 和 SHOW MUTEX STATUS。第 13 章会详细分析它们和其他的命令。

76

2.5.7 不可添加分析代码时的解决办法

When You Can't Add Profiling Code

有时不能添加分析代码或给服务器打补丁，甚至不能改变服务器配置，但是还是有办法进行一些分析。可以试试下面的方法：

- 定制网页服务器的日志，让它们记录挂钟时间和每个请求使用的 CPU 时间。
- 使用数据包嗅探工具在查询经过网络的时候捕获和对查询进行计时（包括网络延迟）。免费的嗅探器包括 mysqlsniffer (<http://hackmysql.com/mysqlsniffer>) 和 tcpdump。请访问 <http://forge.mysql.com/snippets/view.php?id=15> 获得使用 tcpdump 的示例。
- 使用代理，比如 MySQL 代理来捕获和对查询进行计时。

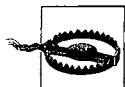
2.6 分析操作系统

Operating System Profiling

通常情况下，检查操作系统的统计数据，并且试图找出操作系统和硬件正在进行的工作是非常有用的技巧。这不仅有助于分析应用程序，而且有助于解决问题。

在此我们侧重于 Unix 类操作系统，因为它最为常见。然而只要其他操作系统提供了统计数据，这些技巧同样也适用。

最常用的工具是 vmstat、iostat、mpstat 和 strace，每个工具都会显示进程、CPU、内存和 I/O 活动等部分不同的信息。大部分 Unix 类操作系统都提供了这些工具。本书展示了如何使用它们的示例，特别是在第 7 章的结尾部分。



警告：注意在基于 GNU/Linux 的产品服务器上使用 strace。它有时和拥有多线程的进程有冲突，并且有可能导致服务器崩溃。

2.6.1 解决 MySQL 连接和进程故障

Troubleshooting MySQL Connections and Processes

我们在别处没有详细讨论的工具是探测网络活动和进行基本故障修复的工具。为了举例，我们将会展示如何追踪 MySQL 连接，回到另外一台服务器的连接发起处。

我们从 SHOW PROCESSLIST 的输出开始，关注进程的 Host 列。使用下面的示例：

```
***** 21. row *****
      Id: 91296
     User: Web
    Host: sargon.cluster3:37636
       db: main
 Command: Sleep
      Time: 10
     State:
      Info: NULL
```

77

Host 列显示连接发起的地方，以及发起的端口。可以使用这个信息找到是哪个进程打开了连接。你如果有使用 root 的权限，可以使用 netstat 和端口号来找到打开连接的进程：

```
root@sargon# netstat -ntp | grep :37636
tcp 0 0 192.168.0.12:37636 192.168.0.21:3306 ESTABLISHED 16072/apache2
```

进程号和名字位于输出的最后一个字段：16072 号进程开启了连接，并且它来自 Apache。一旦知道了进程 ID，就可以顺藤摸瓜，找到其他信息，例如该进程拥有的其他连接：

```
root@sargon# netstat -ntp | grep 16072/apache2
tcp 0 0 192.168.0.12:37636 192.168.0.21:3306 ESTABLISHED 16072/apache2
tcp 0 0 192.168.0.12:37635 192.168.0.21:3306 ESTABLISHED 16072/apache2
tcp 0 0 192.168.0.12:57917 192.168.0.3:389 ESTABLISHED 16072/apache2
```

看上去 Apache 工作进程有两个 MySQL 连接（端口 3306）处于开启状态，并且端口 389 连接在另外一台机器上。端口 389 是什么？这个问题没有确定的回答，但是许多程序都会使用标准的端口号，比如 MySQL 的默认端口是 3306。端口列表通常位于 /etc/services，不妨打开，看看里面怎么说：

```
root@sargon# grep 389 /etc/services
ldap      389/tcp # Lightweight Directory Access Protocol
ldap      389/udp
```

刚好，我们知道该服务器使用了 LDAP 认证，因此这儿使用端口 389 是合理的。再来看看还能发现 16072 号进程的什么信息。可以用 ps 很容易地了解进程正在做的事情。可以按照下面的方式使用 grep，这样就可以在第一行显示列头：

```
root@sargon# ps -eaf | grep 'UID\|16072'
UID      PID PPID C STIME TTY   TIME CMD
apache 16072 22165 0 09:20 ? 00:00:00 /usr/sbin/apache2 -D DEFAULT_VHOST...
```

这些信息提供了查找其他问题的可能性。不要对此感到惊讶，例如 LDAP 或 NFS 这样的服务导致了 Apache 的问题，并且表现为页面的生成速度较慢。

还可以使用 lsof 列出进程打开的文件。这对于查找所有的信息很有帮助，因为 Unix 上面所有的东西都是文件。我们不会在这儿显示它的输出，因为实在太详尽了，但是可以使用 lsof | grep 16072 来找到进程打开的文件。在不能使用 netstat 的时候，还可以使用 lsof 查找网络连接。例如，下面的 lsof 命令显示了和 netstat 大致相同的信息。为了排版需要，我们稍微进行了一下格式化：

```
root@sargon# lsof -i -P | grep 16072
apache2 16072 apache 3u IPv4 25899404 TCP *:80 (LISTEN)
apache2 16072 apache 15u IPv4 33841089 TCP sargon.cluster3:37636->
      hammurabi.cluster3:3306 (ESTABLISHED)
apache2 16072 apache 27u IPv4 33818434 TCP sargon.cluster3:57917->
      romulus.cluster3:389 (ESTABLISHED)
apache2 16072 apache 29u IPv4 33841087 TCP sargon.cluster3:37635->
      hammurabi.cluster3:3306 (ESTABLISHED)
```

在 GNU/Linux 上，/proc 文件系统对于解决问题有极大的帮助。每个进程都在 /proc 下面有自己的文件，并且可以看到关于它的大量信息，比如当前的工作目录、内存使用等。

Apache 实际上有一个类似于 Unix 的 ps 的命令：/server-status/URL。例如，如果局域网在 http://intranet 上运行了 Apache，可以把网页浏览器指向 http://intranet/server-status，以了解 Apache 正在进行的工作。这对于找到进程使用的 URL 是很有帮助的。该页面有一些提示，可以解释输出。

28

2.6.2 高级分析手段和故障解决办法

Advanced Profiling and Troubleshooting



如果需要深入了解进程做的事情——例如，为什么进程处于不可中断的睡眠状态——就可以使用 `strace-p`，以及/或者 `gdb-p`。这些命令能显示系统调用和向后追踪，它能给出进程处于阻塞状态时更多的信息。许多因素都会导致进程阻塞，例如 NFS 锁定了已经崩溃的服务，调用没有响应的远程网页服务等。

也可以更详细地分析系统或部分系统以了解正在进行的工作。如果确实需要高性能并且问题开始出现，那么就有可能要分析 MySQL 内部。尽管这不是你的工作（应该是 MySQL 开发团队的工作，是吧？），但是这有助于隔离导致问题的部分系统。你也许不能或不愿意修复它，但是至少可以设计系统，避开这些暗礁。

下面有一些有用的工具：

OProfile

OProfile (<http://oprofile.sourceforge.net>) 是一个针对 Linux 的系统分析器。它由核心驱动组成并且还有守护进程以收集样例数据，提供一些工具帮助分析收集到的数据。它分析所有的代码，包括终中断句柄、内核、内核模块、应用程序和共享库。如果应用程序是由调试符号编译而成的，OProfile 就能注解源代码，但是这并不是必须的。可以在不重新编译任何东西的情况下分析系统。它的开销相对比较小，大约只有几个百分点。

Gprof

Gprof 是一个 GNU 分析器，它可以对使用了 `-pg` 进行编译的程序产生执行分析代码。它计算了每个函数消耗的时间。Gprof 能够产生方法调用频率和时间的报告，还能产生调用图及注解的代码列表。

另外的工具

另外还有很多工具，包括针对特定程序的工具。这些工具包括 Intel VTune、Sun Performance Analyzer (Sun Studio 的一部分)，以及适用于 Solaris 和其他系统的 DTrace。

架构优化和索引

Schema Optimization and Indexing

优化设计不良或索引不佳的架构 (Schema) 能把性能提高几个数量级。如果需要高性能, 就必须为运行的特定查询设计架构和索引 (Index), 还要评估不同类型查询的性能需求, 因为更改某个查询或架构的一部分会对其他部分造成影响。优化通常需要权衡取舍, 例如, 为了加快数据读取而添加的索引会减慢更新的速度, 同样, 非规范化 (Denormalized) 架构能加快某些类型的查询, 但却会让其他类型的查询变慢。添加计数器和汇总表 (Summary Table) 是优化查询的好方法, 但它们维护的代价很高。

有时必须超越开发人员的身份, 质疑手头的商业需求。通常撰写商业需求的人都不是数据库系统的专家, 他们不会理解这些需求对性能的影响。如果告诉他们一个小小的特性会使硬件需求翻倍, 他们也能理解并不是非得要这个特性。

架构优化和索引 (Indexing) 既需要大局观, 又需要专注于细节。你要了解整个系统, 以弄清楚各部分如何相互影响。本章首先讨论数据类型, 然后讨论覆盖索引 (Covering Index) 策略和规范化。最后则对存储引擎做一些说明。

你有可能在阅读完关于查询优化的章节后再回到本章, 进行回顾。本章讨论的许多主题——尤其是索引——不能被孤立地考虑。你首先要熟悉查询优化及服务器调优, 然后再决定索引。

3.1 选择优化的数据类型

Choosing the Data Type

MySQL 支持很多种不同的数据类型, 并且选择正确的数据类型对于获得高性能至关重要。不管选择何种类型, 下面的简单原则都会有助于做出更好的选择:

更小通常更好

81

一般来说, 要试着使用能正确地存储和表示数据的最小类型。更小的数据类型通常更快, 因为它们使用了更少的磁盘空间、内存和 CPU 缓存, 而且需要的 CPU 周期也更少。

但是要确保不会低估需要保存的值, 在架构中的多个地方增加数据类型的范围是一件极其费时费力的工作。如果不确定需要什么数据类型, 就选择你认为不会超出范围的最小类型。(如果系统不是非常繁忙或不会保存太多的数据, 再或者还处于设计的早期, 就可以在以后轻易地更改它。)

简单就好

越简单的数据类型, 需要的 CPU 周期就越少。例如, 比较整数的代价小于比较字符, 因为字符集和排序规则使字符比较更复杂。这里有两个例子: 一是应该使用 MySQL 内建的类型来保存日期和时间, 而不是

使用字符串；二是应该使用整数来保存 IP 地址。稍后我们再进一步讨论该话题。

尽量避免 NULL

要尽可能地把字段 (Field) 定义为 NOT NULL。即使应用程序无须保存 NULL (没有值)，也有许多表包含了可空列 (Nullable Column)，这仅仅是因为它为默认选项。除非真的要保存 NULL，否则就把列定义为 NOT NULL。

MySQL 难以优化引用了可空列的查询，它会使索引、索引统计和值更加复杂。可空列需要更多的存储空间，还需要在 MySQL 内部进行特殊处理。当可空列被索引的时候，每条记录都需要一个额外的字节，还能导致 MyISAM 中固定大小的索引 (例如一个整数列上的索引) 变成可变大小的索引。

即使要在表中存储“没有值”的字段，还是有可能不使用 NULL 的。考虑使用 0、特殊值或空字符串来代替它。

把 NULL 列改为 NOT NULL 带来的性能提升很小，所以除非确定它引入了问题，否则就不要把它当成优先的优化措施。然后，如果计划对列进行索引，就要尽量避免把它设置为可空。

决定特定列的数据类型的第 1 步就是大致决定数据的类型：数字、字符串、时间等。这通常很直观，但是我们会说到一些不那么直观的特殊情况。

第 2 步是确定特定的类型。许多 MySQL 数据类型能够保存同类的数据，但是存储的范围、精度或物理空间 (磁盘上或内存中) 却不相同。一些数据类型还有特殊的行为或属性。

例如，DATETIME 和 TIMESTAMP 能保存同样类型的数据：日期和时间，精度为秒。然而，TIMESTAMP 使用的空间只有 DATETIME 的一半，还能保存时区，拥有特殊的自动更新能力。而另一方面，它允许的范围要小得多，并且在某些时候，它的特殊功能会成为障碍。

这里我们讨论了基本的数据类型。MySQL 为了保持兼容性，支持很多别名，比如 INTEGER、BOOL 和 NUMERIC。它们都只是别名，它们会让人迷惑，但却不会影响性能。

3.1.1 整数

Whole Number

数字有两种类型：整数 (Whole Number) 和实数 (Real Number)。如果存储整数，就可以使用这几种整数类型：TINYINT、SMALLINT、MEDIUMINT、INT 或 BIGINT，它们分别需要 8、16、24、32 和 64 位存储空间。它们的范围为 $-2^{(N-1)}$ 到 $2^{(N-1)} - 1$ ，这里的 N 是所需存储空间的位数。

整数类型有可选的 UNSIGNED 属性，它表示不允许负数，并大致把正上限提高了一倍。例如，TINYINT UNSIGNED 保存的范围为 0 到 255，而不是 -127 到 128。

有符号 (Signed) 和无符号 (Unsigned) 类型占用的存储空间是一样的，性能也一样。因此可以根据实际情况采用合适的类型。

你的选择将会决定 MySQL 把数据保存在内存中还是磁盘上。然而，整数运算通常使用 64 位 BIGINT 整数，即使是 32 位架构也如此。(一些聚合函数是例外，它们使用 DECIMAL 或 DOUBLE 进行计算。)

MySQL 还可以对整数类型定义宽度，比如 INT(11)。这对于大多数应用程序都是没有意义的：它不会限制值的

范围，只规定了 MySQL 的交互工具（例如命令行客户端）用来显示字符的个数。对于存储和计算，INT(1) 和 INT(20) 是一样的。



提示：Falcon 存储引擎和 MySQL AB 提供的其他存储引擎内部存储整数的机制是不同的。用户不能控制存储数据的实际大小。第三方存储引擎，例如 Brighthouse，也有自己的存储格式和压缩方案。

3.1.2 实数

Real Numbers

83

实数有分数部分。然而，它们并不仅仅是分数。可以使用 DECIMAL 保存比 BIGINT 还大的整数。MySQL 同时支持精确与非精确类型。

FLOAT 和 DOUBLE 类型支持使用标准的浮点运算进行近似计算。如果想知道浮点运算到底如何进行，则要研究平台浮点数的具体实现。

DECIMAL 类型用于保存精确的小数。在 MySQL 5.0 及以上版本，DECIMAL 类型支持精确的数学运算。MySQL 4.1 和早期版本对 DECIMAL 值执行浮点运算，它会因为丢失精度而导致奇怪的结果。在这些 MySQL 版本中，DECIMAL 仅仅是“存储类型”。

在 MySQL 5.0 及以上版本中，服务器自身进行了 DECIMAL 运算，因为 CPU 并不支持对它进行直接计算。浮点运算会快一点，因为计算直接在 CPU 上进行。

可以定义浮点类型和 DECIMAL 类型的精度。对于 DECIMAL 列，可以定义小数点之前和之后的最大位数，这影响了所需的存储空间。MySQL 5.0 和以上版本把数字保存到了一个二进制字符串中（每 4 个字节保存 9 个数字）。例如，DECIMAL(18, 9) 将会在小数点前后都保存 9 位数字，总共使用 9 个字节：小数点前 4 个字节，小数点占 1 个字节，小数点后 4 个字节。

MySQL 5.0 及以上版本中的 DECIMAL 类型最多允许 65 个数字。在较早的版本中，DECIMAL 最多可以有 254 个数字，并且保存为未压缩的字符串（一个数字占一个字节）。然而，这些版本的 MySQL 根本不能在计算中使用如此大的数字，因为 DECIMAL 只是一种存储格式。DECIMAL 在计算时会被转换为 DOUBLE 类型。

可以用多种方式定义浮点数列的精度，它会导致 MySQL 悄悄采用不同的数据类型，或者在保存的时候进行圆整。这些精度定义符不是标准的，因此我们建议定义需要的类型，而不是精度。

比起 DECIMAL 类型，浮点类型保存同样大小的值使用的空间通常更少。FLOAT 占用 4 个字节。DOUBLE 占用 8 个字节，而且精度更高、范围更大。和整数一样，你选择的仅仅是存储类型。MySQL 在内部对浮点类型使用 DOUBLE 进行计算。

由于需要额外的空间和计算开销，只有在需要对小数进行精确计算的时候才使用 DECIMAL，比如保存金融数据。

3.1.3 字符串类型

String Types

84

MySQL 支持很多种字符串类型，它们之间有很多不同。这些数据类型在 MySQL 4.1 和 5.0 中有很大的变化，这

使得它更加复杂。从 MySQL4.1 起，每个字符串列都有自己的字符集和排序规则（更多关于排序规则的内容请参阅第 5 章）。这会极大地影响性能。

VARCHAR 和 CHAR 类型

两种主要的字符串类型是 VARCHAR 和 CHAR。不幸的是，很难确切地解释这两种类型是怎样被保存到磁盘或内存中的，因为具体实现依赖于存储引擎（例如，Falcon 几乎对每种数据类型都用了自己的存储格式）。这里假设你使用的是 InnoDB 和/或 MyISAM。如果不是，请参考所使用的存储引擎的文档。

先看看 VARCHAR 和 CHAR 是如何被保存到磁盘上的。要知道存储引擎可能会使用不同的方式把 CHAR 和 VARCHAR 类型保存到内存中，并且服务器从存储引擎取回这些值的时候还可能会把它转换为其他存储格式。下面是两种类型的比较：

VARCHAR

VARCHAR 保存了可变长度的字符串，是使用得最多的字符串类型。它能比固定长度类型占用更少的存储空间，因为它只占用了自己需要的空间（也就是说较短的值占用的空间就较少）。例外情况是使用 ROW_FORMAT=FIXED 创建的 MyISAM 表，它为每行使用固定长度的空间，可能会造成浪费。

VARCHAR 使用额外的 1 到 2 字节来存储值的长度。如果列的最大长度小于或等于 255，则使用 1 字节，否则就使用 2 字节。假设使用 latin1 字符集，VARCHAR(10) 将会占用 11 字节的存储空间。VARCHAR(1000) 则会占用 1002 字节，因为需要 2 个字节来保存长度信息。

VARCHAR 能节约空间，所以对性能有帮助。然而，由于行的长度是可变的，它们在更新的时候可能会发生变化，这会引起额外的工作。如果行的长度增加并不再适合于原始的位置时，具体的行为则会和存储引擎相关。例如，MyISAM 会把行拆开，InnoDB 则可能进行分页。另外的存储引擎还可能不会在合适的位置更新数据。

当最大长度远大于平均长度，并且很少发生更新的时候，通常适合使用 VARCHAR。这时候碎片就不会成为问题。还有当你使用复杂的字符集，比如 UTF-8 时，它的每个字符都可能会占用不同的存储空间。

5 在 5.0 及以上版本，无论是保存还是取值，MySQL 都会保留字符串末尾的空格。但是在 4.1 及之前的版本，这些空格会被去掉。

CHAR

CHAR 是固定长度的。MySQL 总是为特定数量的字符分配足够的空间。当保存 CHAR 值的时候，MySQL 会去掉任何末尾的空格。（在 MySQL4.1 及之前版本，VARCHAR 也是如此。CHAR 和 VARCHAR 在逻辑上是一样的，只是存储格式不同。）进行比较的时候，空格会被填充到字符串末尾。

CHAR 在存储很短的字符串或长度近似相同的字符串的时候很有用。例如，CHAR 适合用存储用户密码的 MD5 哈希值，它的长度总是一样的。对于经常改变的值，CHAR 也好于 VARCHAR，因为固定长度的行不容易产生碎片。对于很短的列，CHAR 的效率也高于 VARCHAR。CHAR(1) 字符串对于单字节字符集（注 1）只会占用 1 个字节，但是 VARCHAR(1) 则会占用 2 个字节，因为有 1 个字节用来存储长度信息。

这种行为可能会让人有一点点迷惑，这儿有个例子。首先，先创建一个只有 1 列（CHAR(10)）的表，并向里面

注 1：要记住长度是用字符表示的，而不是字节。一个多字节字符集的每个字符占用的空间都可能会超过 1 个字节。

保存一些信息：

```
mysql> CREATE TABLE char_test( char_col CHAR(10));
mysql> INSERT INTO char_test(char_col) VALUES
-> ('string1'), (' string2'), ('string3 ');
```

从表中取值的时候，字符串末尾的空格已经被删除了：

```
mysql> SELECT CONCAT("'", char_col, "'") FROM char_test;
+-----+
| CONCAT("'", char_col, "'") |
+-----+
| 'string1'                  |
| ' string2'                  |
| 'string3'                  |
+-----+
```

如果以 VARCHAR(10) 的方式保存，就会得到下面的结果：

```
mysql> SELECT CONCAT("'", varchar_col, "'") FROM varchar_test;
+-----+
| CONCAT("'", varchar_col, "'") |
+-----+
| 'string1'                      |
| ' string2'                      |
| 'string3 '                      |
+-----+
```

数据如何保存取决于存储引擎，并非所有的存储引擎都会按照相同的方式来处理定长和可变长度的字符串。Memory 存储引擎使用了固定长度的行，因此当它面对可变长度字段的时候就会分配可能的最大空间。而 Falcon 引擎即使是对固定长度的 CHAR 字段，也会使用长度可变的列。但是，填充和截取空格的行为在各个存储引擎之间都是一样的，因为这是 MySQL 服务器自身的行为。

CHAR 和 VARCHAR 的兄弟类型为 BINARY 和 VARBINARY，它们用于保存二进制字符串。二进制字符串和传统的字符串很类似，但是它们保存的是字节，而不是字符。填充也有所不同：MySQL 使用 \0 (0 字节) 填充 BINARY 值，而不是空格，并且不会在获取数据的时候把填充的值截掉（注 2）。

它们在须要存储二进制数据并且想让 MySQL 按照字节进行比较的时候是有用的。字节比较的优势并不仅仅体现在大小写敏感上。MySQL 按照字节的数值进行比较，比按字符比较简单得多，效率也更高。

慷慨是不明智的

使用 VARCHAR(5) 和 VARCHAR(200) 保存 'hello' 占用的空间都是一样的，那么使用较短的列有任何优势吗？

其实有巨大的优势。较大的列会使用更多的内存，因为 MySQL 通常会分配固定大小的内存块来保存值。这对排序或使用基于内存的临时表尤其不好。同样的事情也会发生在使用文件排序或基于磁盘的临时表的时候。

最好的策略就是只分配真正需要的空间。

注 2：如果不想在取值的时候，值发生改变，就不要使用 BINARY 类型，因为 MySQL 会用 \0 把它填充到需要的长度。

BLOB 和 TEXT 类型

BLOB 和 TEXT 分别以二进制和字符形式保存大量数据。

事实上,它们各自有自己的数据类型家族:字符类型有 TINYTEXT、SMALLTEXT、TEXT、MEDIUMTEXT 和 LONGTEXT,二进制类型有 TINYBLOB、SMALLBLOB、BLOB、MEDIUMBLOB 和 LONGBLOB。BLOB 等同于 SMALLBLOB, TEXT 等同于 SMALLTEXT。

8 和其他类型不同,MySQL 把 BLOB 和 TEXT 当成有实体的对象来处理。存储引擎通常会特别地保存它们。InnoDB 在它们较大的时候会使用单独的“外部”存储区域来进行保存。每个值在行里面都需要 1 到 4 字节,并且还需要足够的外部存储空间来保存实际的值。

BLOB 和 TEXT 唯一的区别就是 BLOB 保存的是二进制数据,没有字符集和排序规则,但是 TEXT 有字符集和排序规则。

MySQL 对 BLOB 和 TEXT 列的排序方式和其他类型不同:它不会按照字符串的完整长度进行排序,而只是按照 max_sort_length 规定的前若干个字节进行排序。如果只按照开始的几个字符排序,就可以减少 max_sort_length 的值或使用 ORDER BY SUBSTRING(column, length)。

MySQL 不能索引这些数据类型的完整长度,也不能为排序使用索引。(后文有这方面的更多内容。)

如何避免磁盘上的临时表

由于 Memory 存储引擎不支持 BLOB 和 TEXT 类型,使用了 BLOB 和 TEXT 列并且需要隐式临时表的查询将不得不使用磁盘上的 MyISAM 临时表,即使只有几列也会这样。这会导致严重的性能开销。即使把 MySQL 配置为使用 RAM 磁盘上的临时表,也需要很多昂贵的操作系统调用。(Maria 存储引擎会把所有的东西都缓存到内存中,有助于减轻这个问题。)

最好的办法是尽可能地避免使用 BLOB 和 TEXT 类型。如果不能避免,就可以使用 ORDER BY SUBSTRING(column, length)把这些值转换为字符串,让它们使用内存中的临时表。要保证使用的子字符串足够短,不要让临时表变得过大,以致超过 max_heap_size 或 tmp_table_size 的大小,否则 MySQL 会把表转换为磁盘上的 MyISAM 表。

如果在解释器 (EXPLAIN) 中的 Extra 列显示“使用临时表 (Using Temporary)”,就说明使用了隐式临时表。

使用 ENUM 代替字符串类型

有时可以使用 ENUM 列来代替传统的字符串类型。ENUM 列可以存储 65 535 个不同的字符串。MySQL 以非常紧凑的方式保存了它们,根据列表中值的数量,MySQL 会把它们压缩到 1 到 2 个字节中。MySQL 在内部把每个值都保存为整数,以表示值在列表中的位置,并且还保留了一份“查找表 (Lookup Table)”来表示整数和字符串在表的 .frm 文件中的映射关系。下面有一个例子:

```
88 mysql> CREATE TABLE enum_test(  
    ->     e ENUM('fish', 'apple', 'dog') NOT NULL  
    -> );  
mysql> INSERT INTO enum_test(e) VALUES('fish'), ('dog'), ('apple');
```

这 3 行实际存储的是整数,而不是字符串。可以把它们按照数字取出来,这样可以了解它们的双重特性:


```
mysql> SELECT e + 0 FROM enum_test;
+-----+
| e + 0 |
+-----+
|      1 |
|      3 |
|      2 |
+-----+
```

如果用数字定义 ENUM 常量，这种双重性会非常迷惑人。我们建议不要这么做。

另外一个让人吃惊的事情是 ENUM 字段在内部是按数字顺序进行排序的，而不是按字符串排序：

```
mysql> SELECT e FROM enum_test ORDER BY e;
+-----+
| e      |
+-----+
| fish   |
| apple  |
| dog    |
+-----+
```

绕过这个问题的方法就是按照想要的顺序来定义 ENUM。还可以使用 FIELD() 显式地规定排序顺序，但这会造成排序无法使用索引。

```
mysql> SELECT e FROM enum_test ORDER BY FIELD(e, 'apple', 'dog', 'fish');
+-----+
| e      |
+-----+
| apple  |
| dog    |
| fish   |
+-----+
```

ENUM 最不好的一面是字符串列表是固定的，并且添加或删除字符串须使用 ALTER TABLE。因此，对于一系列未来可能会改变的字符串，使用 ENUM 就不是一个好主意。MySQL 在内部的权限表中使用 ENUM 来保存 Y 值和 N 值（译注 1）。

由于 MySQL 把每个值保存为整数，并且须进行查找才能把它转换为字符串表示，所以 ENUM 列有一些开销。这通常可以由它们较小的大小进行弥补，但并不总是这样。在特定情况下，把 CHAR 或 VARCHAR 列和 ENUM 列进行联接，可能会比联接另一个 CHAR 或 VARCHAR 列慢。

为了说明该结论，我们对一个应用中的一张表进行了基准测试，看看在 MySQL 中执行这样一个联接的速度有多快。该表有一个很宽的主键：

```
CREATE TABLE Webservicecalls (
  day date NOT NULL,
  account smallint NOT NULL,
  service varchar(10) NOT NULL,
  method varchar(50) NOT NULL,
  calls int NOT NULL,
  items int NOT NULL,
  time float NOT NULL,
  cost decimal(9,5) NOT NULL,
  updated datetime,
  PRIMARY KEY (day, account, service, method)
) ENGINE=InnoDB;
```

译注 1：Yes 和 No。

表包含了 11 万行数据，只占用了 10MB 空间，因此它可以完全被装入内存中。service 列包含了 5 个不同的值，平均长度为 4 字符，并且 method 列包含了 71 个值，平均长度为 20 个字符。

拷贝该表并把 service 和 method 列转换为 ENUM，如下：

```
CREATE TABLE Webservicecalls_enum (
  ... omitted ...
  service ENUM(...values omitted...) NOT NULL,
  method ENUM(...values omitted...) NOT NULL,
  ... omitted ...
) ENGINE=InnoDB;
```

然后用主键列联接这两个表，查询如下：

```
mysql> SELECT SQL_NO_CACHE COUNT(*)
-> FROM Webservicecalls
-> JOIN Webservicecalls USING(day, account, service, method);
```

我们用 VARCHAR 和 ENUM 的不同组合测试了这个查询，结果如表 3-1 所示。

表 3-1：联接 VARCHAR 和 ENUM 列的速度

| 测试 | 每秒查询数 |
|--------------------|-------|
| VARCHAR 联接 VARCHAR | 2.6 |
| VARCHAR 联接 ENUM | 1.7 |
| ENUM 联接 VARCHAR | 1.8 |
| ENUM 联接 ENUM | 3.5 |

把列转换为 ENUM 后，联接会变快，但是 VARCHAR 联接 ENUM 会较慢。在本例中，既然它们不一定非要以 VARCHAR 方式进行联接，那么转换这些列就是一个好主意。

然而，转换列还有另外一个好处。根据 SHOW TABLE STATUS 命令的 Data_length 列的结果，把这两列转换为 ENUM 会把表缩小 1/3。在某些情况下，即使会出现 ENUM 和 VARCHAR 联接，这也是值得的。同样，转换后主键自己也只有一半大小了。因为这是 InnoDB 表，如果表上有另外的索引，减小主键的大小会使它们变得更小。稍后再解释该问题。

3.1.4 日期和时间类型

Date and Time Types

MySQL 可以使用多种类型来保存各种日期和时间值，比如 YEAR 和 DATE。MySQL 能存储的最细的时间粒度是秒。然而，它可以用毫秒的粒度进行暂时的运算。我们会说明如何绕过这种存储限制。

大部分临时类型都没有替代者，因此没有什么是什么是最佳选择的问题。唯一的问题是保存日期和时间的时候需要什么。MySQL 提供了两种相似的数据类型：DATETIME 和 TIMESTAMP。对于很多应用程序，它们都能工作，但是在某些情况下，一种会好于另外一种。

DATETIME

这个类型能保存大范围的值，从 1001 年到 9999 年，精度为秒。它把日期和时间封装到一个格式为 YYYYMMDDHHMMSS 的整数中，与时区无关。它使用了 8 字节存储空间。

在默认情况下，MySQL 以一种可排序的、清楚的格式显示 DATETIME 值，例如 2008-01-16 22:37:08。这种表示日期和时间的方式符合 ANSI 标准。

TIMESTAMP

就像它的名字一样，TIMESTAMP 类型保持了自 1970 年 1 月 1 日午夜（格林尼治标准时间）以来的秒数，它和 Unix 的时间戳相同。TIMESTAMP 只使用了 4 字节的存储空间，因此它的范围比 DATETIME 小得多。它表示只能从 1970 年到 2038 年。MySQL 提供了 FROM_UNIXTIME() 函数把 Unix 时间戳转换为日期，并提供了 UNIX_TIMESTAMP() 函数，把日期转换为 Unix 时间戳。

较新的 MySQL 版本按照 DATETIME 格式化 TIMESTAMP 的值，但是较老的 MySQL 不会在各个部分之间显示任何标点符号。这仅仅是显示上的区别，TIMESTAMP 存储格式在所有版本中都是一样的。

TIMESTAMP 显示的值也依赖于时区。MySQL 服务器、操作系统及客户端连接都有时区设置。

因此，保存 0 值的 TIMESTAMP 实际显示为美国东部时间 1969-12-31 19:00:00，与格林尼治标准时间 (GMT) 相差 5 小时。

TIMESTAMP 也有 DATETIME 没有的特殊性质。在默认情况下，如果插入的行没有定义 TIMESTAMP 列的值，MySQL 就会把它设置为当前时间（注 3）。在更新的时候，如果没有显式地定义 TIMESTAMP 列的值，MySQL 也会自动更新它。可以配置 TIMESTAMP 列的插入和更新行为。最后，TIMESTAMP 列默认是 NOT NULL，这和其他的数据类型都不一样。

除了这些特殊行为之外，通常应该使用 TIMESTAMP，因为它比 DATETIME 更节约空间。有时人们把 Unix 的时间戳保存为整数，但是这通常没有任何好处。因为这种格式处理起来不太方便，我们并不推荐它。

如果需要以秒以下的精度保存日期和时间，应该怎么办？MySQL 当前没有适合的数据类型，但是可以使用自己的存储格式：可以使用 BIGINT 类型并且把它以毫秒的精度保存为时间戳格式，或者使用 DOUBLE 保存秒的小数部分。这两种办法都不错。

3.1.5 位集数据类型 (Bit-Packed Data Types)

Bit-Packed Data Types

MySQL 有几种数据类型可以利用值里面的二进制位来紧凑地保存数据。无论下面的何种存储方式和操作，所有这些数据类型从技术上来说都是字符串类型。

BIT

在 MySQL5.0 之前，BIT 仅仅是 TINYINT 的同名词。但是在 MySQL5.0 及以上版本中，它成了完全不同的、有特殊性质的新类型。

可以使用 BIT 列，并且在列中保存一个或多个 true/false 值。BIT(1) 定义了一个只含有一个二进制位的字段，BIT(2) 能保存 2 位，以此类推。BIT 列的最大长度是 64 位。

BIT 的行为在不同存储引擎之间是不同的。MyISAM 出于存储的目的，会把列包起来，因此 17 个单独的

注 3：TIMESTAMP 的行为规则很复杂，并且随着 MySQL 版本的变化而变化，因此应该验证它的行为。在更改了 TIMESTAMP 列之后检查 SHOW CREATE TABLE 的输出通常是个好主意。

BIT 列只需要 17 位存储空间（假设所有列都不允许 NULL）。MyISAM 会把存储空间圆整为 3 个字节。另外的存储引擎，例如 Memory 和 InnoDB，把每个列保存为最小的整数类型已经足够容纳所有位了，因此不会节约任何存储空间。

MySQL 把 BIT 当成字符串类型，而不是数字类型。当获取一个 BIT(1) 值的时候，结果是一个字符串，但是内容是二进制的 0 或 1，而不是 ASCII 值 “0” 或 “1”。然而，如果以数字的方式把它取出来，结果就是该二进制字符串转换出来的数字。例如，如果把值 b'00111001'（等于 57）保存到 BIT(8) 列中，并且取出来，这时候就会得到包含了字符代码为 57 的字符串，它正好等于 ASCII 字符 “9”。但如果用数字的方式，就会得到 57：

```
mysql> CREATE TABLE bittest(a bit(8));
mysql> INSERT INTO bittest VALUES(b'00111001');
mysql> SELECT a, a + 0 FROM bittest;
+-----+-----+
| a      | a + 0 |
+-----+-----+
| 9      | 57     |
+-----+-----+
```

这相当迷惑人，因此我们推荐小心使用 BIT。对于大多数程序，最好能避免使用该类型。

如果只想保存一位的 true/false，另外一个选择是创建一个可空的 CHAR(0) 列。它能保存空值（NULL）或 0 长度的值（空字符串）。

SET

如果要保存许多 true/false 值，可以考虑把许多列合并为 SET 数据类型，它在 MySQL 内部是以一系列位表示的。它有效地使用了存储空间，并且 MySQL 有诸如 FIND_IN_SET() 和 FIELD() 这样的函数，方便在查询中使用它。它的主要缺点是改变列定义的代价较高，需要 ALTER TABLE，这在大型表上是很昂贵的操作（但是本章稍后部分给出了解决办法）。通常说来，你也不能在 SET 列上使用索引进行查找。

在整数列上进行按位操作

SET 的一种替代方式是利用整数包装一系列位。例如，可以把 8 位包装到 TINYINT 中，并且利用按位操作符来操纵它们。可以在应用程序中为每个位定义命名常量来简化这个工作。

比起 SET，这种办法主要的好处在于可以不使用 ALTER TABLE 而改变字段代表的“枚举”。缺点就是查询比较难写，也比较难懂（当第 5 位被设置的时候表示什么意思？）。一些人习惯按位操作，但是另外一些人不会，因此是否采用这种办法取决于个人的喜好。

一个使用位集的应用程序的例子就是用来保存许可的访问控制表（ACL, Access Control List）。每一位或 SET 元素代表了 CAN_READ、CAN_WRITE 或 CAN_DELETE 值。如果使用了 SET 列，就可以让 MySQL 在列定义中保存位到值的映射。如果使用的是整数，就要在应用程序中保存这种映射关系。下面是使用 SET 列的查询：

```
mysql> CREATE TABLE acl (
->   perms SET('CAN_READ', 'CAN_WRITE', 'CAN_DELETE') NOT NULL
-> );
mysql> INSERT INTO acl(perms) VALUES ('CAN_READ,CAN_DELETE');
mysql> SELECT perms FROM acl WHERE FIND_IN_SET('CAN_READ', perms);
+-----+
| perms                |
+-----+
| CAN_READ,CAN_DELETE |
+-----+
```

如果使用整数，例子就会像这样：

```
mysql> SET @CAN_READ := 1 << 0,
->      @CAN_WRITE := 1 << 1,
->      @CAN_DELETE := 1 << 2;
mysql> CREATE TABLE acl (
-> perms TINYINT UNSIGNED NOT NULL DEFAULT 0
-> );
mysql> INSERT INTO acl(perms) VALUES (@CAN_READ + @CAN_DELETE);
mysql> SELECT perms FROM acl WHERE perms & @CAN_READ;
+-----+
| perms |
+-----+
|      5 |
+-----+
```

我们使用了变量来定义值，但是在程序中使用常量来代替。

3.1.6 选择标识符

(Identifier Column)

为标识符列 (Identifier Column) 选择好的数据类型是非常重要的。你可能会更多地用它们和其他列做比较，还可能把它们用作其他表的外键，因此为标识符列选择数据类型的时候，你也可能是在为相关的表选择数据类型。（正如本章前文所述，最好在相关的表中使用同样的数据类型，因为有可能会对它们执行联接。）

当为标识符列选择数据类型的时候，不仅要考虑存储类型，还要考虑 MySQL 如何对它们进行计算和比较。例如，MySQL 在内部把 ENUM 和 SET 类型保存为整数，但是在比较的时候把它们转换为字符串。

一旦选择了数据类型，要确保在相关表中使用同样的类型。类型之间要精确匹配，包括诸如 UNSIGNED 这样的属性（注 4）。混合不同的数据类型会导致性能问题，即使没有性能问题，隐式的类型转换也能导致难以察觉的错误。在你已经忘记了自己是在对不同类型做比较的时候，这些错误就会突然出现。

选择最小的数据类型能表明所需值的范围，并且为将来留出增长空间。例如，如果用 state_id 表示美国的州，这时不会有成千上万个值，因此不必使用 INT。TINYINT 就足够了，它比 INT 小 3 字节。如果把它用作外键，就会导致较大的性能差距。

整数类型

整数通常是标识符的最佳选择，因为它速度快，并且能使用 AUTO_INCREMENT。

ENUM 和 SET

ENUM 和 SET 通常不适合用作标识符，尽管它适合用来做静态的，包含了状态和“类型”值的“定义表”。ENUM 和 SET 列适合用来保存订单的状态、产品的类别或性别这样的信息。

举个例子，如果使用 ENUM 字段来定义产品的类别，那么对同一个 ENUM 字段，就会需要一个查找表。（可以给这个查找表添加描述性文本、产生词汇表，或者在网站的下拉菜单中提供有意义的文字。）在这个例

注 4：如果使用了 InnoDB 存储引擎，就不能创建类型不精确匹配的外键。它的错误信息是：“ERROR 1005 (HY000): Can't create table”。该信息在某些情况下会造成困惑，在 MySQL 邮件列表上经常会出现关于它的问题。（奇怪的是，可以在不同长度的 VARCHAR 之间创建外键。）

子中，可以把 ENUM 用作标识符，但是对于大多数情况都应该避免这么做。

字符串类型

要尽可能地避免使用字符串来做标识符，因为它们占用了很多空间并且通常比整数类型要慢。特别注意不要在 MyISAM 表上使用字符串标识符。MyISAM 默认情况下为字符串使用了压缩索引 (Packed Index)，这使查找更为缓慢。在我们的测试中，使用了压缩索引的 MyISAM 表性能要慢 6 倍。

还要特别注意的是完全“随机”的字符串，例如由 MD5()、SHA1() 或 UUID() 产生的。它们产生的每一个新值都会被任意地保存在很大的空间范围内，这会减慢 INSERT 及一些 SELECT 查询 (注 5)。

- 它们会减慢 INSERT 查询，因为插入的值会被随机地放入索引中。这会导致分页、随机磁盘访问及聚集存储引擎上的聚集索引碎片。
- 它们会减慢 SELECT 查询，因为逻辑上相邻的行会分布在磁盘和内存中的各个地方。
- 随机值导致缓存对所有类型的查询性能都很差，因为它们会使缓存赖以工作的访问局部性 (Locality of Reference) 失效。如果整个数据集都变得同样“热”的时候，那么把特定部分的数据缓存到内存中就没有任何的优势了，并且如果工作集不能被装入内存中，缓存就会进行很多刷写的工作，并且会导致很多缓存未命中。

如果保存 UUID 值，就应该移除其中的短横线，更好的办法是使用 UHEX() 把 UUID 值转化为 16 字节的数字，并把它保存在 BINARY(16) 列中。可以使用 HEX(0) 函数以十六进制数的格式把值提取出来。

使用 UUID() 产生的值和其他哈希函数得到的值比起来，有特殊的性质：UUID 值不是均匀分布的，并且有一定程度的顺序性。但它还是比不上单调递增的整数。

3.1.7 特殊类型的数据

Special Types of Data

一些数据没有直接对应的内建数据类型。精度低于秒的时间戳就是一个例子，本章前面已经叙述了如何保存这种数据。

另外一个例子是 IP 地址。人们通常使用 VARCHAR(15) 列来保存 IP 地址。但是，IP 地址实际是无符号的 32 位整数，而不是字符串。使用小数点来进行分隔纯粹是为了增加它的可读性。实际应该使用无符号整数来保存 IP 地址。MySQL 提供了 INET_ATON() 和 INET_NTOA() 函数在 IP 地址和整数之间相互转换。未来的 MySQL 也许会提供原生的 IP 地址数据类型。

3.2 索引基础知识

Indexing Basics

索引 (Index) 是帮助 MySQL 高效获取数据的数据结构。它对于高性能非常关键，但是人们通常会忘记或误解

注 5：另一方面，对于一些进行很多写入操作的大型表来说，这些伪随机的值实际能减少“数据热点” (Hot Spot (译注 2))。

译注 2：指容易出错的数据。

它，因此建立索引（Indexing）是现实中性能问题的首要原因。这是我们把它放在本书前面的原因，甚至比讨论查询优化还要靠前。

索引（MySQL 中也叫“键（Key）”）在数据越大的时候越重要。规模小、负载轻的数据库即使没有索引，也能有好的性能，但是当数据增加的时候，性能就会很快下降。理解索引如何工作的最简单的方式就是把索引看成一本书。为了找到书中一个特定的话题，你须要查看目录，它会告诉你页码。

小心自动生成的架构

我们已经讨论了最重要的数据类型（某些对性能有严重影响，某些的影响相对较小），但是还没有说到自动生成的架构的危害。

写得不好的架构迁移程序和自动生成架构的程序能导致严重的性能问题。一些程序为所有字段使用巨大的 VARCHAR 列，或者在联接中使用不同类型的列。如果架构是自动生成的，一定要仔细检查它。

对象关系映射（Object Relational Mapping, ORM）系统（以及使用它的“框架”）是另外一个常见的性能噩梦。一些系统可以让你在后端以任何格式保存任何数据，这通常意味着它们并没有使用数据类型的能力。有时它们把每个对象的每个属性都保存在不同的行中，甚至还会使用基于时间戳的版本控制，因此每个属性都有多个版本！

这种设计对开发人员有吸引力，因为这能使他们以面向对象的方式工作，而不用考虑数据如何存储。然而，这种“向开发人员隐藏了复杂性”的应用程序通常没有很好的扩展性。我们建议仔细衡量用性能交换生产率的想法，并且总是用实际的大型数据集进行测试，这样就不会在很晚的时候才发现性能问题。

MySQL 按类似的方式使用索引。它会搜索索引的数据结构来找到值。当发现匹配的时候，它就会包含这个匹配的行。假设运行下面的查询：

```
mysql> SELECT first_name FROM sakila.actor WHERE actor_id = 5;
```

索引列位于 actor_id 列，因此 MySQL 会使用索引找到 actor_id 为 5 的行。换句话说，它在索引中按值进行了查找并且返回任何包含该值的行。

索引包含了来自于表中某一列或多个列的值。如果索引了多列数据，那么列的顺序非常重要，因为 MySQL 只能高效地搜索索引的最左前缀（Leftmost Prefix）。如你所见，创建一个双列索引和两个单列索引是不一样的。

3.2.1 索引类型

Types of Indexes

有很多类型的索引，它们各有自己的性能特点。索引是在存储引擎层实现的，而不是服务器层。因此，它们并不是标准化的：每个引擎的索引工作方式略有不同，并不是所有的引擎都支持所有类型的索引。即使多个引擎支持同样的索引，它们的实现方式也可能有所不同。

现在就来看看 MySQL 当前支持的索引类型，以及它们的优点和缺点。

B-Tree 索引

当人们谈及索引而没有说明其类型的时候，多半是指 B-Tree 索引，它通常使用 B-Tree 数据结构来保存数据（注 6）。大部分 MySQL 的存储引擎都支持这种索引。Archive 引擎是个例外：它直到 MySQL 5.1 才支持索引，而且只支持索引单个 AUTO_INCREMENT 列。

使用“B-Tree”来称呼这种索引的原因是 MySQL 在 CREATE TABLE 和其他命令中使用了它。然而，存储引擎可能会在内部使用不同的存储结构。例如，NDB Cluster 存储引擎尽管把索引标记为 BTREE，但是内部使用的是 T-Tree 数据结构。

存储引擎使用了不同的方式把索引保存到磁盘上，它们会影响性能。例如，MyISAM 使用前缀压缩（Prefix Compression）以减小索引，而 InnoDB 不会压缩索引，因为它不能把压缩索引用于某些优化。同样，MyISAM 索引按照行存储的物理位置引用被索引的行，但是 InnoDB 按照主键值引用行。这些不同有各自的优点和缺点。

B-Tree 通常意味着数据存储是有序的，并且每个叶子页（Leaf Page）到根的距离是一样的。图 3-1 显示了 B-Tree 索引的抽象表示，它大致显示了 InnoDB 的索引工作过程（InnoDB 使用了 B+Tree 结构）。MyISAM 使用了不同的结构，但是原则是类似的。

B-Tree 索引加速了数据访问，因为存储引擎不会扫描整个表得到需要的数据。相反，它从根节点开始（图中没有显示）。根节点保存了指向子节点的指针，并且存储引擎会根据指针寻找数据。它通过查找节点页中的值找到正确的指针，节点页包含子节点中值的上界和下界。最后，存储引擎可能无法找到需要的数据，也可能成功地找到包含数据的叶子页面（Leaf Page）。

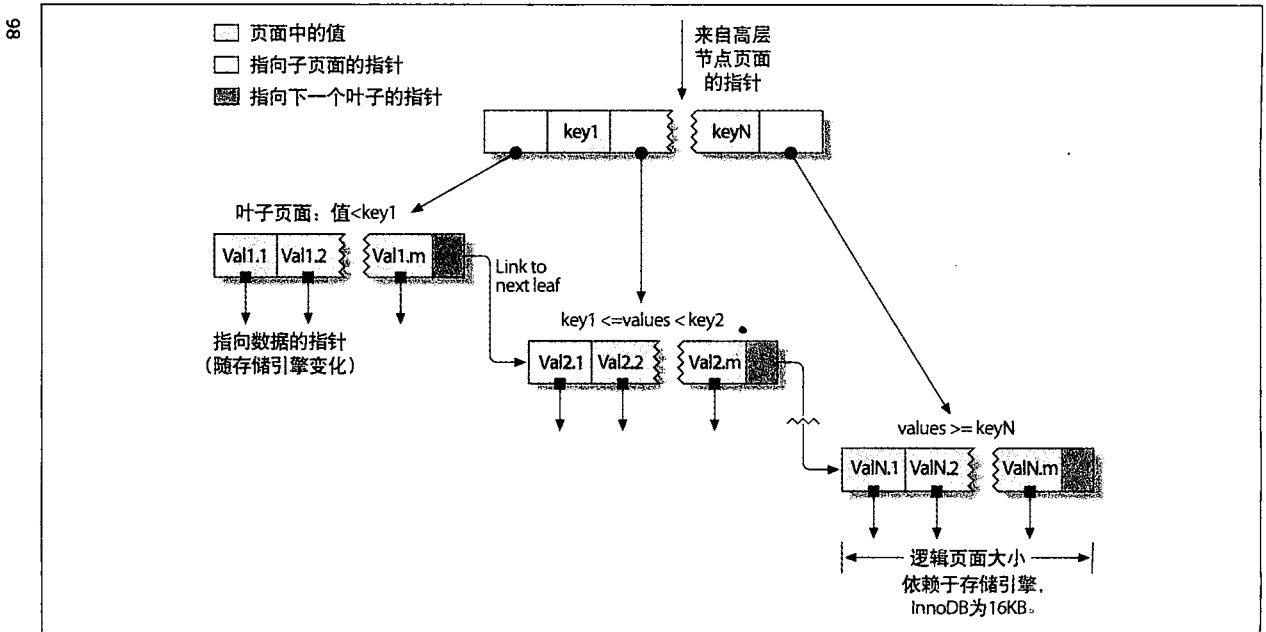


图 3-1：建立在 B-Tree（从技术上是 B+Tree）结构上的索引

叶子页面是很特殊的，因为它们含有指向被索引的数据的指针，而没有指向其他页面的指针。（不同的存储引擎

注 6：许多存储引擎实际使用了 B+Tree 索引，它的每一个叶子节点都包含了指向下一个节点的联接，以实现快速的范围遍历。可以参考计算机学术文献以了解 B-Tree 索引的细节。

有不同类型的指向数据的“指针”。) 图 3-1 只显示了一个节点页面和它的叶子页面，但是在根节点和叶子之间可能有很多层节点页面。树的深度取决于表的大小。

因为 B-Tree 按顺序保存了索引的列，它们对于搜索范围数据很有用。例如，一个文本字段的索引会按照字母顺序在树上依次排列，因此查找“名字开头字母从 I 到 K 的人”的效率会很高。

假设有下面的表：

```
CREATE TABLE People (
  last_name  varchar(50)    not null,
  first_name varchar(50)    not null,
  dob        date           not null,
  gender     enum('m', 'f') not null,
  key(last_name, first_name, dob)
);
```

对于表中的每一行数据，索引将会包含 last_name、first_name 和 dob 列。图 3-2 显示了索引如何安排它保存的数据。

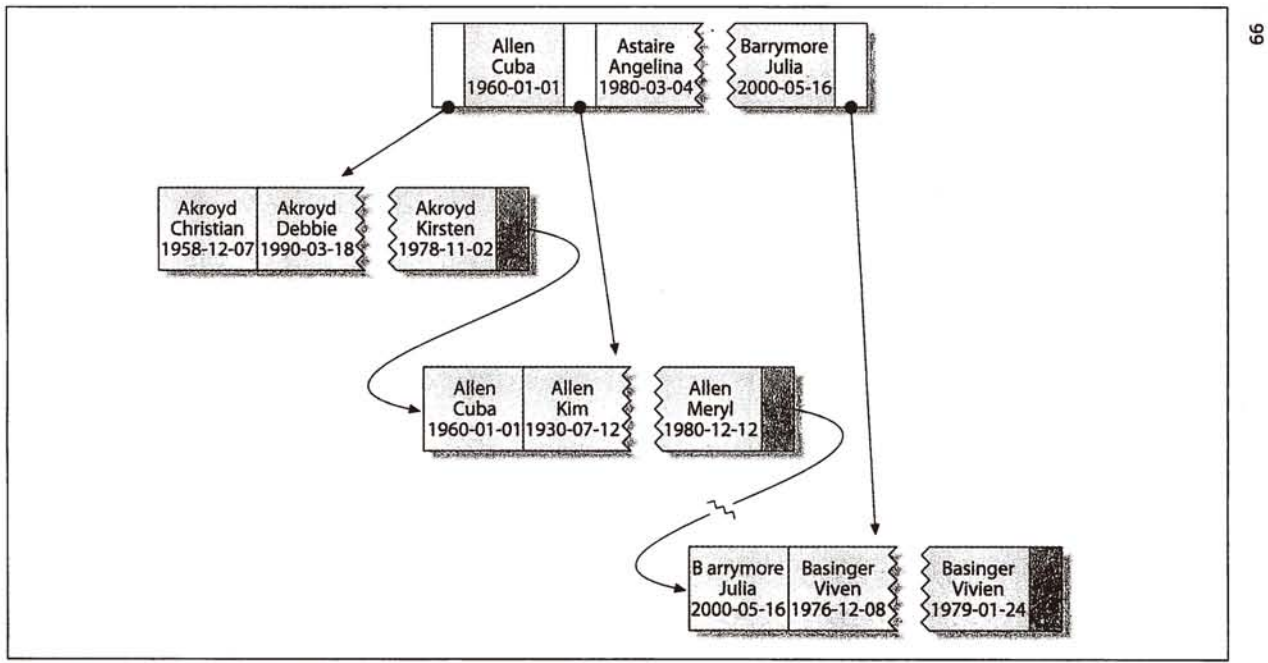


图 3-2：B-Tree（技术上是 B+Tree）中数据示例

注意到索引保存数据的顺序等于 CREATE TABLE 命令中列的顺序。看看最后两条记录：有两个人名字相同，但是出生日期不同，他们是按照出生日期排序的。

能使用 B-Tree 索引的查询类型。B-Tree 索引能很好地用于全键值、键值范围或键前缀查找。它们只有在查找使用了索引的最左前缀（Leftmost Prefix）的时候才有用（注 7）。上节中的索引对于以下类型的查询有用。

注 7：这和 MySQL 特定相关，甚至和版本特定相关。另外的数据库可以使用非前导索引部分（nonleading index parts），尽管通常使用完全前缀（complete prefix）更有效率。MySQL 也许会在以后提供该选项，本章稍后会说明该问题的解决办法。

匹配全名

全键值匹配指和索引中的所有列匹配。例如，索引可以帮你找到一个叫 Cuba Allen 并且出生于 1960-01-01 的人。

匹配最左前缀

B-Tree 索引可以帮你找到姓为 Allen 的所有人。这仅仅适用了索引中的第一列。

匹配列前缀

可以匹配某列的值的开头部分。这种索引能帮你找到所有姓氏以 J 开头的人。这只会使用索引的第 1 列。

匹配范围值

这种索引能帮你找到姓大于 Allen 并且小于 Barrymore 的人。这也只会使用索引第一列。

精确匹配一部分并且匹配某个范围中的另一部分

这种索引能帮你找到姓为 Allen 并且名字以字母 K (Kim、Karl 等) 开头的人。它精确匹配了 last_name 列并且对 first_name 列进行了范围查询。

只访问索引的查询

B-Tree 索引通常能支持只访问索引的查询，它不会访问数据行。第 120 页“覆盖索引”讨论了这种优化。

由于树的节点是排好序的，它们可以用于查找（查找值）和 ORDER BY 查询（以排序的方式查找值）。通常来说，如果 B-Tree 能以某种特殊的方式找到某行，那么它也能以同样的方式对行进行排序。因此，上面讨论的所有查找方式也可以同等地应用于 ORDER BY。

下面是 B-Tree 索引的一些局限：

- 如果查找没有从索引列的最左边开始，它就没什么用处。例如，这种索引不能帮你找到所有叫 Bill 的人，也不能找到所有出生在某天的人，因为这些列不在索引的最左边。同样，你不能使用该索引查找某个姓氏以特定字符结尾的人。
- 不能跳过索引中的列。也就是说，不能找到所有姓氏为 Smith 并且出生在某个特定日期的人。如果不确定 first_name 列的值，MySQL 就只能使用索引的第一列。
- 存储引擎不能优化访问任何在第一个范围条件右边的列。比如，如果查询是 WHERE last_name='Smith' AND first_name LIKE 'J%' AND dob='1976-12-23'，访问就只能使用索引的头两列，因为 LIKE 是范围条件（但是服务器能把其余列用于其他目的）。对于某个只有有限值的列，通常使用等于条件，而不是范围条件来绕过这个问题。本章稍后的索引案例中我们会举出详细的例子。

现在你应该知道为什么列顺序极端重要了：这些局限都和列顺序有关。对于高性能应用程序，也许要针对相同列以不同顺序创建多个索引，以满足程序要求。

一些局限并不是 B-Tree 固有的，而是 MySQL 查询优化器和存储引擎使用索引的方式造成的。其中一些局限以后将会被去掉。

哈希索引

哈希索引 (Hash Index) 建立在哈希表的基础上，它只对使用了索引中的每一列的精确查找有用 (注 8)。对于每一行，存储引擎计算出了被索引列的哈希码 (Hash Code)，它是一个较小的值，并且有可能和其他行的哈希码不同。它把哈希码保存在索引中，并且保存了一个指向哈希表中每一行的指针。

在 MySQL 中，只有 Memory 存储引擎支持显式的哈希索引。尽管 Memory 表也有 B-Tree 索引，但它是 Memory 表的默认索引类型。Memory 引擎支持不唯一的哈希索引，它在数据库系统中并不常见。如果多个值有相同的哈希码，索引就会把行指针以链表 (Linked List) 的方式保存在哈希表的同一条记录中。

下面有个例子，假设有下表：

```
CREATE TABLE testhash (  
  fname VARCHAR(50) NOT NULL,  
  lname VARCHAR(50) NOT NULL,  
  KEY USING HASH(fname)  
) ENGINE=MEMORY;
```

包含了下面的数据：

```
mysql> SELECT * FROM testhash;  
+-----+-----+  
| fname | lname |  
+-----+-----+  
| Arjen | Lentz |  
| Baron | Schwartz |  
| Peter | Zaitsev |  
| Vadim | Tkachenko |  
+-----+-----+
```

现在假设索引使用一个假想的哈希函数，叫 f()，它返回下面的值 (仅仅是例子，不是真实的值)：

```
f('Arjen') = 2323  
f('Baron') = 7437  
f('Peter') = 8784  
f('Vadim') = 2458
```

索引的数据结构会像这样：

| 数据片段 (Slot) | 值 (Value) |
|-------------|-------------|
| 2323 | 指向 row1 的指针 |
| 2458 | 指向 row2 的指针 |
| 7437 | 指向 row3 的指针 |
| 8784 | 指向 row4 的指针 |

注意到数据片段是排好序的，但数据行不是。现在，当执行下面查询的时候：

```
mysql> SELECT lname FROM testhash WHERE fname='Peter';
```

MySQL 会计算 'Peter' 的哈希值并且使用它去查找索引中的指针。因为 f('Peter')=8784，MySQL 就会在索引中查找 8784 并找到指向第 3 行的指针。最后的步骤是用 Peter 和第 3 行中的数据进行比较，确定它是不是正确的数据行。

注 8：更多哈希表的细节，请参考计算机科学文献。

因为索引本身只保存简短的哈希值，哈希索引显得非常紧凑。哈希值的长度不会依赖于索引的列，TINYINT 列的哈希索引和大型字符列的哈希索引大小是一样的。

这样看来，查找的速度是很快的。然而，哈希索引有一些局限：

- 因为索引只包含了哈希码和行指针，而不是值自身，MySQL 不能使用索引中的值来避免读取行。幸运的是，访问内存中的行很快，因此这通常不会降低性能。
- MySQL 不能使用哈希索引进行排序，因为它们不会按序保存行。
- 哈希索引不支持部分键匹配，因为它们是由被索引的全部值计算出来的。也就是说，如果在 (A, B) 两列上有索引，并且 WHERE 子句中只使用了 A，那么索引就不会起作用。
- 哈希索引只支持使用了 =、IN() 和 <=> 的相等比较（注意 <> 和 <=> 不是相同的运算符）。它们不能加快范围查询，例如 WHERE price > 100。
- 访问哈希索引中的数据非常快，除非碰撞率很高（很多值有相同的哈希码）。当发生碰撞的时候，存储引擎必须访问链表中的每一个行指针，然后逐行进行数据比较，以确定正确的数据。
- 如果有很多碰撞，一些索引维护操作就有可能变慢。如果在一个选择性很低（很多碰撞值）的列上创建哈希索引，然后从表中删除一行，那么从索引中找到行的代价会很高。存储引擎将不得不检查哈希链表中的每一行，以找到和移除被删除行的索引。

这些限制使哈希索引只在特殊情况下有用。然而，当它们符合应用程序需要的时候，就能极大地改善性能。一个例子就是数据仓库应用程序，其中经典的“星形”架构要对查找表进行很多联接操作。哈希索引正是查找表所需要的。

除了 Memory 存储引擎的显式哈希索引，NDB Cluster 存储引擎支持唯一的哈希索引。它的功能是该存储引擎特有的，本书不会进行讨论。

InnoDB 存储引擎有一个特别的功能，叫自适应哈希索引 (Adaptive Hash Index)。当 InnoDB 注意到一些索引值被很频繁地访问的时候，它就会在 B-Tree 的顶端为这些值建立起内存中的索引。这使 B-Tree 索引有了一些哈希索引的特性，例如很快地哈希查找。这个过程是全自动的，既不能控制，也不能配置它。

建立自己的哈希索引。如果存储引擎不支持哈希索引，就可以按照 InnoDB 使用的方式模拟自己的哈希索引。这会让你得到某些哈希索引的特定，例如很大的键也只有很小的索引。

想法非常简单：在标准 B-Tree 索引上创建一个伪哈希索引。它和真正的哈希索引不是一回事，因为它还是使用 B-Tree 索引进行查找。然而，它将会使用键的哈希值进行查找，而不是键自身。你所要做的事情就是在 WHERE 子句中手动地定义哈希函数。

一个不错的例子就是 URL 查找。URL 通常会导致 B-Tree 索引变大，因为它们非常长。通常会按照下面的方式来查找 URL 表：

```
mysql> SELECT id FROM url WHERE url="http://www.mysql.com";
```

但是，如果移除掉 url 列上的索引并且给表添加一个被索引的 url_crc 列，就可以按照下面的方式进行查询：

```
mysql> SELECT id FROM url WHERE url="http://www.mysql.com"
-> AND url_crc=CRC32("http://www.mysql.com);
```

这种方式很不错，因为 MySQL 查询优化器注意到 `url_crc` 列上有很小的、选择性很高的索引，并且它会使用里面的值进行索引查找（本例中是 1 560 514 994）。即使有几列相同的 `url_crc` 值，也很容易进行精确的对比来确定需要的行。替代方案是把完整 URL 索引为字符串，它要慢得多。

这个办法的一个缺点就是要维护哈希值。你可以手工进行维护，在 MySQL 5.0 及以上版本中，可以使用触发器来进行维护。下面的例子显示了触发器如何在插入和更新值的时候维护 `url_crc` 列。首先，创建一个表：

```
CREATE TABLE pseudohash (
  id int unsigned NOT NULL auto_increment,
  url varchar(255) NOT NULL,
  url_crc int unsigned NOT NULL DEFAULT 0,
  PRIMARY KEY(id)
);
```

接下来创建触发器。我们先暂时更改一下命令分隔符，这样就可以在触发器中使用分号：

```
DELIMITER |

CREATE TRIGGER pseudohash_crc_ins BEFORE INSERT ON pseudohash FOR EACH ROW BEGIN
SET NEW.url_crc=crc32(NEW.url);
END;
|

CREATE TRIGGER pseudohash_crc_upd BEFORE UPDATE ON pseudohash FOR EACH ROW BEGIN
SET NEW.url_crc=crc32(NEW.url);
END;
|

DELIMITER ;
```

剩下的工作就是验证触发器自动维护了哈希值：

```
mysql> INSERT INTO pseudohash (url) VALUES ('http://www.mysql.com');
mysql> SELECT * FROM pseudohash;
+-----+-----+-----+
| id | url | url_crc |
+-----+-----+-----+
| 1 | http://www.mysql.com | 1560514994 |
+-----+-----+-----+
mysql> UPDATE pseudohash SET url='http://www.mysql.com/' WHERE id=1;
mysql> SELECT * FROM pseudohash;
+-----+-----+-----+
| id | url | url_crc |
+-----+-----+-----+
| 1 | http://www.mysql.com/ | 1558250469 |
+-----+-----+-----+
```

如果使用这种方式，就不应该使用 `SHA1()` 和 `MD5()` 这些哈希函数。它们返回很长的字符串，会浪费大量的存储空间并且减慢比较速度。它们是强加密函数，被设计为不产生任何冲突。这并不是我们的目标。简单的哈希函数能在有较好性能的同时保证可接受的冲突率。

如果表有很多行并且 `CRC32()` 产生了很多冲突，就要实现自己的 64 位哈希函数。要确保自己的函数返回整数，而不是字符串。一种实现 64 位哈希函数的方法是利用 `MD5` 返回的部分值。这可能会比自己写的用户自定义函数（User Defined Function）效率要低一些（参阅第 230 页“用户自定义函数”），但是在紧急关头它还是能派上用场的。

```
mysql> SELECT CONV(RIGHT(MD5('http://www.mysql.com/'), 16), 16, 10) AS HASH64;
+-----+
| HASH64 |
+-----+
| 9761173720318281581 |
+-----+
```

Maatkit (<http://maatkit.sourceforge.net>) 包含了一个用户自定函数 (UDF)，实现了 Fowler/Noll/Vo64 位哈希算法，速度非常快。

处理哈希碰撞。当通过哈希值搜索值的时候，必须在 WHERE 子句中包含一个常量值 (Literal Value)：

```
mysql> SELECT id FROM url WHERE url_crc=CRC32("http://www.mysql.com")
-> AND url="http://www.mysql.com";
```

下面的查询不能正常工作，因为如果有另外一个 URL 的 CRC32 () 值也是 1 560 514 994，查询将会返回 2 行：

```
mysql> SELECT id FROM url WHERE url_crc=CRC32("http://www.mysql.com");
```

由于所谓的生日悖论 (Birthday Paradox)，哈希碰撞几率的增长比你想象的要快。CRC32 () 返回一个 32 位的整数值，因此至少需要 93 000 个值才能达到碰撞概率为 1%。出于演示的目的，我们把/usr/share/dict/words 中的所有单词加载到了一个表中，一共有 98 569 行，同时生成了它们的 CRC32 () 值。现在已经有哈希冲突了！碰撞使下面的查询返回的结果多于 1 行：

```
mysql> SELECT word, crc FROM words WHERE crc = CRC32('gnu');
+-----+-----+
| word | crc |
+-----+-----+
| coddng | 1774765869 |
| gnu | 1774765869 |
+-----+-----+
```

正确的查询是：

```
mysql> SELECT word, crc FROM words WHERE crc = CRC32('gnu') AND word = 'gnu';
+-----+-----+
| word | crc |
+-----+-----+
| gnu | 1774765869 |
+-----+-----+
```

106 为了避免碰撞问题，必须在 WHERE 子句中定义两个条件。如果碰撞不是问题，比如进行统计并且不需要精确的结果，就可以通过在 WHERE 子句中使用 CRC32 () 值简化查询，并得到效率提升。

空间 (R-Tree) 索引

MyISAM 支持空间索引 (Spatial Index)，它可以使用诸如 GEOMETRY 这样的地理空间数据类型 (Geospatial Type)。和 B-Tree 索引不同，空间索引不会要求 WHERE 子句使用索引的最左前缀。它同时全方位地索引了数据。这样就可以高效地使用任何数据组合进行查找。然而，必须使用 MySQL GIS 函数，例如 MBRCONTAINS ()，才能得到这个好处。

全文索引

FULLTEXT 是 MyISAM 表的一种特殊索引。它从文本中找到关键字，而不是直接和索引中的值进行比较。全文

搜索和其他匹配类型完全不同。它有很多微妙的地方，比如停用词（Stopword）、词干和复数（Stemming and Plurals）、布尔搜索（Boolean Searching）。它包含的工作远不止存储引擎进行简单的 WHERE 参数匹配。

在某列上有全文索引不能磨灭在同一列上有 B-Tree 索引的价值。全文索引用于 MATCH AGAINST 操作，而不是普通的 WHERE 子句操作。

第 244 页“全文搜索”详细讨论了全文索引的细节。

3.3 高性能索引策略

Indexing Strategies for High Performance

创建正确的索引并恰当地使用它们，是获得高性能的必要条件。我们已经介绍了不同类型的索引并且探究了它们的优缺点。现在来看看如何开发索引的能力。

有效地选择和使用索引的方式是多种多样的，因为有很多特殊的优化行为和特别的行为。决定在何时采取什么样的行动，并且衡量它们对性能的影响是一直要学习的技能。以下各节有助于理解如何有效地使用索引，但是不要忘记测试！

3.3.1 隔离列

Isolate the Column

如果在查询中没有隔离索引的列，MySQL 通常不会使用索引。“隔离”列意味着它不是表达式的一部分，也没有位于函数中。

例如，下面的查询不能使用 actor_id 上的索引：

```
mysql> SELECT actor_id FROM sakila.actor WHERE actor_id + 1 = 5;
```

人们能轻易地看出 WHERE 子句中的 actor_id 等于 4，但是 MySQL 却不会帮你求解方程，这取决于自己。你应该养成简化 WHERE 子句的习惯，这样就会把被索引的列单独放在比较运算符的一边。

下面是另外一种常见的错误：

```
mysql> SELECT ... WHERE TO_DAYS(CURRENT_DATE) - TO_DAYS(date_col) <= 10;
```

这个查询将会查找 date_col 值离今天不超过 10 天的所有行，但是它不会使用索引，因为使用了 TO_DAYS() 函数。下面是一种较好的方式：

```
mysql> SELECT ... WHERE date_col >= DATE_SUB(CURRENT_DATE, INTERVAL 10 DAY);
```

这个查询就可以使用索引，但是还可以改进。使用 CURRENT_DATE 将会阻止查询缓存把结果缓存起来，可以用常量替换掉 CURRENT_DATE 的值：

```
mysql> SELECT ... WHERE date_col >= DATE_SUB('2008-01-17', INTERVAL 10 DAY);
```

查询缓存的细节请参阅第 5 章。

3.3.2 前缀索引和索引选择性

Prefix Index and Index Selectivity

有时需要索引很长的字符列，它会使索引变大并且变慢。一个策略就是模拟哈希索引。但是有时这也不好，那么应该怎么办呢？

通常可以索引开始的几个字符，而不是全部值，以节约空间并得到好的性能。这使索引需要的空间变小，但是也会降低选择性。索引选择性 (Index Selectivity) 是不重复的索引值 (也叫基数 (Cardinality)) 和表 (#T) 中所有行的比值，它的值在 $1/\#T$ 和 1 之间。高选择性的索引有好处，因为它使 MySQL 在查找匹配的时候可以过滤掉更多的行。唯一索引的选择率为 1，为最佳值。

列前缀通常可以提供高性能所需的足够选择性。如果索引 BLOB 和 TEXT 列，或者很长的 VARCHAR 列，就必须定义前缀索引，因为 MySQL 不允许索引它们的全文。

矛盾在于选择足够长的前缀会得到好的选择性，但是短的前缀会节约空间。前缀应该足够长，使它的选择性能够接近于索引整个列。换句话说，前缀的基数性应该接近于全列的基数性。

为了决定好的前缀长度，须找到最常见的值，以及最常见的前缀进行对比。Sakila 数据库没有表能很好地展示这种想法，因此我们从 city 表衍生了一个表，这样就有足够的数据进行演示：

108

```
CREATE TABLE sakila.city_demo(city VARCHAR(50) NOT NULL);
INSERT INTO sakila.city_demo(city) SELECT city FROM sakila.city;
-- Repeat the next statement five times:
INSERT INTO sakila.city_demo(city) SELECT city FROM sakila.city_demo;
-- Now randomize the distribution (inefficiently but conveniently):
UPDATE sakila.city_demo
SET city = (SELECT city FROM sakila.city ORDER BY RAND() LIMIT 1);
```

现在有了一个示例数据集。结果不是真实的，另外我们使用了 RAND()，所以你的结果可能会有所不同，但是对于练习并不重要。首先，我们找到最常见的城市：

```
mysql> SELECT COUNT(*) AS cnt, city
-> FROM sakila.city_demo GROUP BY city ORDER BY cnt DESC LIMIT 10;
+-----+-----+
| cnt | city          |
+-----+-----+
| 65  | London       |
| 49  | Hiroshima    |
| 48  | Teboksary    |
| 48  | Pak Kret     |
| 48  | Yaound       |
| 47  | Tel Aviv-Jaffa |
| 47  | Shimoga      |
| 45  | Cabuyao      |
| 45  | Callao       |
| 45  | Bislig       |
+-----+-----+
```

注意每个值都出现了 45 到 65 次。现在查找最常见的名字前缀，取前 3 个字母：

```
mysql> SELECT COUNT(*) AS cnt, LEFT(city, 3) AS pref
-> FROM sakila.city_demo GROUP BY pref ORDER BY cnt DESC LIMIT 10;
+-----+-----+
| cnt | pref |
+-----+-----+
```

| | |
|-----|-----|
| 483 | San |
| 195 | Cha |
| 177 | Tan |
| 167 | Sou |
| 163 | al- |
| 163 | Sal |
| 146 | Shi |
| 136 | Hal |
| 130 | Val |
| 129 | Bat |

每个前缀出现的次数更多了，因此唯一前缀比唯一城市名少得多。这时的想法就是增加前缀长度，直到它的选择性接近于列全长的选择性。试验证明前缀长度为 7 比较合适：

```
mysql> SELECT COUNT(*) AS cnt, LEFT(city, 7) AS pref
-> FROM sakila.city_demo GROUP BY pref ORDER BY cnt DESC LIMIT 10;
```

| cnt | pref |
|-----|---------|
| 70 | Santiag |
| 68 | San Fel |
| 65 | London |
| 61 | Valle d |
| 49 | Hiroshi |
| 48 | Teboksa |
| 48 | Pak Kre |
| 48 | Yaound |
| 47 | Tel Avi |
| 47 | Shimoga |

109

计算合适前缀长度的另外一种办法就是计算全列的选择性，并使前缀的选择性接近于它。下面显示了如何找到全列的选择性：

```
mysql> SELECT COUNT(DISTINCT city)/COUNT(*) FROM sakila.city_demo;
```

| COUNT(DISTINCT city)/COUNT(*) |
|-------------------------------|
| 0.0312 |

平均来说，如果前缀的选择率能接近 0.31，基本就可以了。可以在同一个查询中针对许多不同的前缀长度进行计算，这对于大表非常有用。下面是如何在一个查询中计算不同前缀长度的选择率：

```
mysql> SELECT COUNT(DISTINCT LEFT(city, 3))/COUNT(*) AS sel3,
-> COUNT(DISTINCT LEFT(city, 4))/COUNT(*) AS sel4,
-> COUNT(DISTINCT LEFT(city, 5))/COUNT(*) AS sel5,
-> COUNT(DISTINCT LEFT(city, 6))/COUNT(*) AS sel6,
-> COUNT(DISTINCT LEFT(city, 7))/COUNT(*) AS sel7
-> FROM sakila.city_demo;
```

| sel3 | sel4 | sel5 | sel6 | sel7 |
|--------|--------|--------|--------|--------|
| 0.0239 | 0.0293 | 0.0305 | 0.0309 | 0.0310 |

查询显示当长度接近于 7 的时候，选择率提升的幅度就会减少。

只看平均选择率是不够的。还要考虑最坏情况下的选择率。平均选择率会让你觉得长度 4 或 5 的前缀就足够了，

但是，如果数据分布非常不均匀，这可能就会有问题。如果使用长度为 4 的前缀查看出现最多城市的次数，就能明显看出不均匀性：

```
110 mysql> SELECT COUNT(*) AS cnt, LEFT(city, 4) AS pref
      -> FROM sakila.city_demo GROUP BY pref ORDER BY cnt DESC LIMIT 5;
+-----+-----+
| cnt | pref |
+-----+-----+
| 205 | San  |
| 200 | Sant |
| 135 | Sout |
| 104 | Chan |
| 91  | Toul |
+-----+-----+
```

使用 4 个字符，最常见的索引出现的次数比最常见的城市全名要多得多。这就是说，它的选择性低于平均选择性。如果有比这个随机生成的示例更真实的数据，就更有可能看见这种现象。例如，在真实的城市名上创建 4 个字符的前缀索引，对于以“San”和“New”打头的城市来说，选择率会很差，因为有很多城市都以这几个字母开头。

现在已经找到了合适的前缀长度，下面是如何创建前缀索引的：

```
mysql> ALTER TABLE sakila.city_demo ADD KEY (city(7));
```

前缀索引能很好地减少索引的大小及提高速度，但是它也有坏处：MySQL 不能在 ORDER BY 或 GROUP BY 查询中使用前缀索引，也不能把它们用作覆盖索引（Covering Index）。



提示：有时后缀索引（Suffix Index）也挺有用（例如，查找某个域名的所有电子邮件地址）。MySQL 不支持反向索引（Reversed Index），但是可以把反向字符串保存起来，并且索引它的前缀。可以用触发器维护这种索引。具体细节请参阅本章 103 页“建立自己的哈希索引”。

3.3.3 聚集索引

Created by TopSage

聚集索引（注 9）（Clustered Index）不是一种单独的索引类型，而是一种存储数据的方式。其具体细节依赖于实现方式，但是 InnoDB 的聚集索引实际上在同样的结构中保存了 B-Tree 索引和数据行。

当表有聚集索引的时候，它的数据行实际保存在索引的叶子页（Leaf Page）中。术语“聚集”指实际的数据行和相关的键值都保存在一起（注 10）。每个表只能有一个聚集索引，因为不能一次把行保存在两个地方。（但是，覆盖索引可以模拟多个聚集索引，本章稍后会谈到。）

111 由于是存储引擎负责实现索引，因此不是所有的存储引擎都支持聚集索引。当前，SolidDB 和 InnoDB 是唯一支持聚集索引的存储引擎。本节重点讨论 InnoDB，但是这儿讨论的原则对于任何支持聚集索引的存储引擎都基本适用。

图 3-3 显示了聚集索引中的记录是如何存放的。注意到叶子页包含了行的全部数据，但是节点页只含有被索引

注 9：Oracle 用户熟悉“索引组织表（index-organized table）”，它们是同样的概念。

注 10：稍后你会看到，并不总是这样。

的列。在这个例子中，被索引的列包含的是整数。

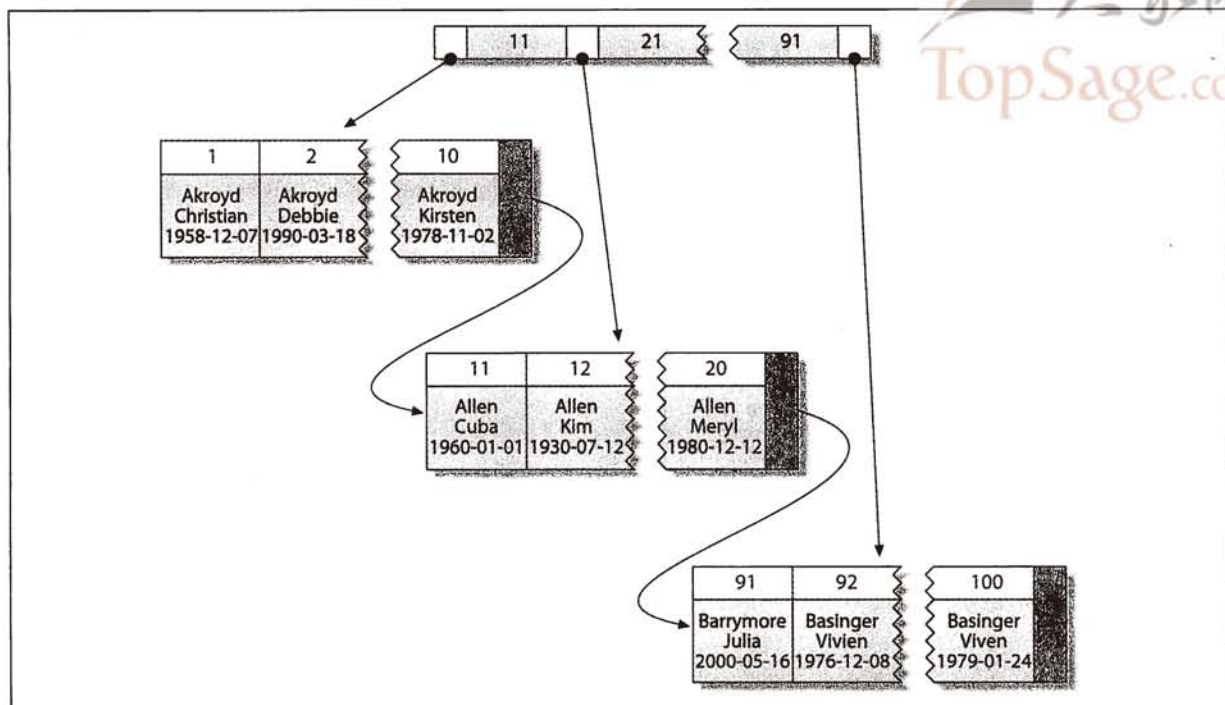


图 3-3：聚集索引数据布局

一些数据库服务器可以选择聚集的列，但是写作本书的时候，没有任何 MySQL 的存储引擎能做到这点。InnoDB 按照主键（Primary Key）进行聚集。这意味着图 3-3 中“被索引的列”其实是主键列。

如果没有定义主键，InnoDB 会试着使用唯一的非空索引（Unique Nonnullable Index）来代替。如果没有这种索引，InnoDB 就会定义隐藏的主键然后上面进行聚集（注 11）。InnoDB 只聚集在同一页面中的记录。包含相邻键值的页面也许会相距甚远。

聚集主键有助于性能，但是它也能导致严重的性能问题。因此应该仔细思索聚集，特别是把表的存储引擎在 InnoDB 和其他引擎互换的时候。

聚集的数据有如下优点：

- 可以把相关数据保存在一起。例如，当实现电子邮箱的时候，可以按照 `user_id` 进行聚集，这样从磁盘上提取几个页面的数据就能把某个用户的邮件全部抓取出来。如果没有使用聚集，读取每个邮件都会访问磁盘。
- 数据访问快。聚集索引把索引和数据都保存到了同一棵 B-Tree 中，因此从聚集索引中取得数据通常比在非聚集索引进行查找要快。
- 使用覆盖索引的查询可以使用包含在叶子节点中的主键值。

如果表和查询可以使用它们，这些优点能极大地提高性能。然而，聚集索引也有缺点：

注 11：SolidDB 引擎也会这么做。

- 聚集能最大限度地提升 I/O 密集负载的性能。如果数据能装入内存，那么其顺序也就无所谓了，这样聚集就没什么用处。
- 插入速度严重依赖于插入顺序。按照主键的顺序插入行是把数据装入 InnoDB 表最快的方法。如果没有按照主键顺序插入数据，那么在插入之后最好使用 OPTIMIZE TABLE 重新组织一下表。
- 更新聚集索引是昂贵的，因为它强制 InnoDB 把每个更新的行移到新的位置。
- 建立在聚集索引上的表在插入新行，或者在行的主键被更新，该行必须被移动的时候会进行分页（Page Split）。分页发生在行的键值要求行必须被放到一个已经放满了数据的页的时候，此时存储引擎必须分页才能容纳该行。分页会导致表占用更多的磁盘空间。
- 聚集表可能会比全表扫描慢，尤其在表存储得比较稀疏或因为分页而没有顺序存储的时候。
- 第二（非聚集）索引可能会比预想的大，因为它们的叶子节点包含了被引用行的主键列。
- 第二索引访问需要两次索引查找，而不是一次。

最后一点有点迷惑人。为什么第二索引需要两次索引查找？答案就在于第二索引保存的“行指针”的本质。记住，叶子节点不会保存引用的行的物理位置，而是保持了行的主键值。

113 这意味着为了从第二索引查找行，存储引擎首先要找到叶子节点，然后使用保存在那里的主键值找到主键，最终找到行。这需要两次动作，两次 B-Tree 导航，而不是一次。（在 InnoDB 中，自适应哈希索引能减少这种损失。）

比较 InnoDB 和 MyISAM 的数据布局

聚集和非聚集数据布局的区别，以及相应的主索引和第二索引的区别，会使人迷惑并出乎人们的预料。看看 InnoDB 和 MyISAM 如何对下面的表进行布局：

```
CREATE TABLE layout_test (
  col1 int NOT NULL,
  col2 int NOT NULL,
  PRIMARY KEY(col1),
  KEY(col2)
);
```

假定该表的主键值从 1 增加到 10 000，随机地进行插入，然后使用 OPTIMIZE TABLE 优化。换句话说，数据在磁盘上的分布已经优化过了，但是行的顺序可能是随机的。Col2 的值在 1 到 100 之间随机进行分配，这样就会产生很多重复的数据。

MyISAM 的数据布局。MyISAM 的布局要简单一些，所以首先介绍它。MyISAM 按照插入的顺序把值保存在磁盘上，如图 3-4 所示。

行号显示在行外面，从 0 开始。由于行的尺寸是固定的，MyISAM 能从表开头跳过所需的字节找到需要的行。（MyISAM 并不总是使用上图中的“行号”，它会根据行的大小是固定还是可变的，选择不同的策略。）

| 行号 | col1 | col2 |
|------|------|------|
| 0 | 99 | 8 |
| 1 | 12 | 56 |
| 2 | 3000 | 62 |
| ... | | |
| 9997 | 18 | 8 |
| 9998 | 4700 | 13 |
| 9999 | 3 | 93 |

图 3-4: MyISAM 中表 layout_test 的数据布局

这种布局很容易建立索引。下面显示的一系列图表，隐藏了页面的物理细节，只显示索引中的“节点”。索引中的每个叶子节点包含了行号。图 3-5 显示了表的主键。

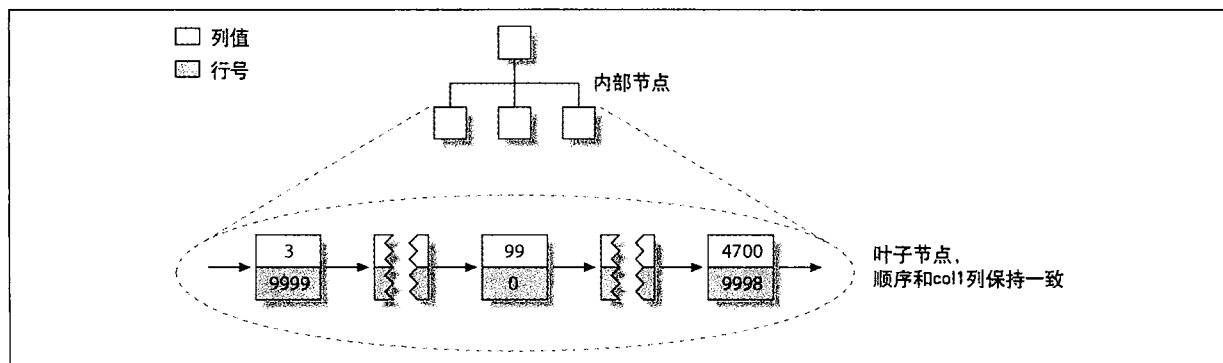


图 3-5: MyISAM 中 layout_test 表的主键布局

我们已经忽略了一些细节，比如前一个 B-Tree 节点内部有多少子节点，但是这对于理解非聚集存储引擎的基本数据布局并不重要。

col2 列上的索引又会如何？它有什么特殊的吗？回答是否定的，它和其他索引没什么区别。图 3-6 显示了 col2 列上的索引。

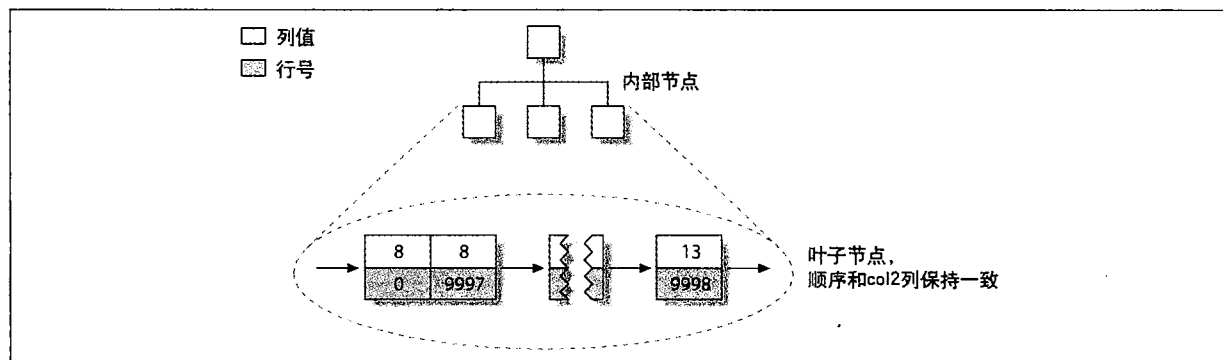


图 3-6: MyISAM 中 layout_text 表的 col2 列索引布局

事实上，MyISAM 中主键和其他索引没什么结构上的区别。主键只是一个唯一的，名为 PRIMARY 的非空索引。

InnoDB 的数据布局。InnoDB 支持聚集索引，因此它以很不一样的方式保存相同数据。InnoDB 按图 3-7 的方式

存储数据:

115

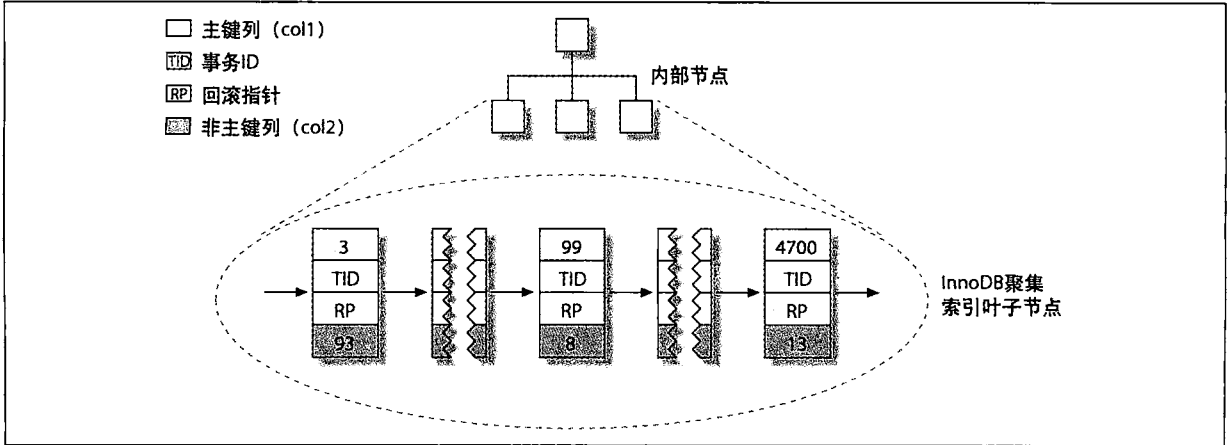


图 3-7: InnoDB 中 layout_text 表的主键布局

第一眼看上去, 和图 3-5 没什么不同。但是再看就会注意到该图显示了整个表, 而不是只有索引。由于聚集索引“是”InnoDB 中的表, 所以不会像 MyISAM 那样需要单独的行存储空间。

聚集索引中的每个叶子节点都包含主键值、事务 ID、用于事务和 MVCC 的回滚指针及剩下的列 (本例中是 col2)。如果主键被创建在列的前缀上, InnoDB 也会包含整个列及剩下的所有列。

还有一个和 MyISAM 不同的是, InnoDB 中的第二索引和聚集索引很不一样。InnoDB 的第二索引叶子节点包含了主键值作为指向行的“指针”, 而不是“行指针”。这种策略减少了在移动行或数据分页的时候索引的维护工作。使用行的主键值作为指针使得索引变得更大, 但是这意味着 InnoDB 可以移动行, 而无须更新指针。

图 3-8 显示了示例表的 col2 索引。

每个叶子节点包含了被索引的列 (本例只有 col2), 然后是主键值 (col1)。

这些图形显示了 B-Tree 叶子节点, 但是我们故意忽略了非叶子节点。InnoDB 的非叶子节点包含了被索引的列, 外加一个指向下一节点的指针 (下一节点可以是非叶子节点, 也可以是叶子节点)。这适用于聚集索引和第二索引。

116

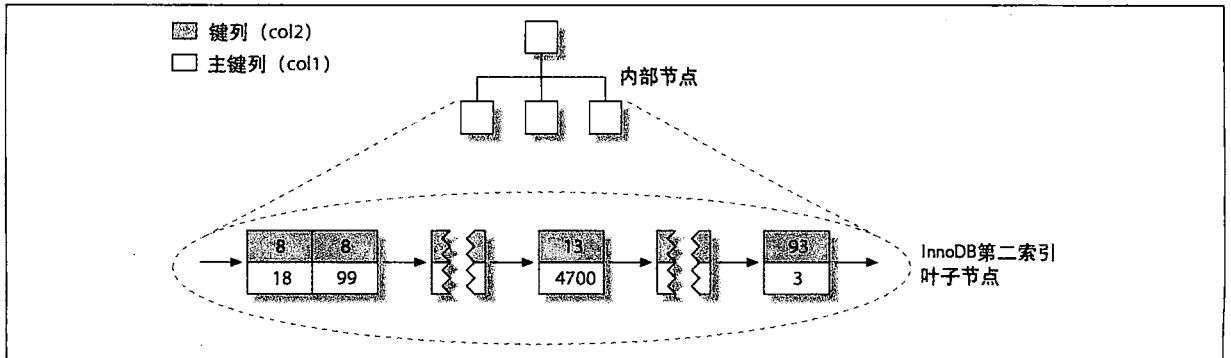


图 3-8: InnoDB 中 layout_text 表的第二索引布局

图 3-9 是 InnoDB 和 MyISAM 安排表的示意图。从图中可以轻易地看出 InnoDB 和 MyISAM 保存数据和索引的区别。

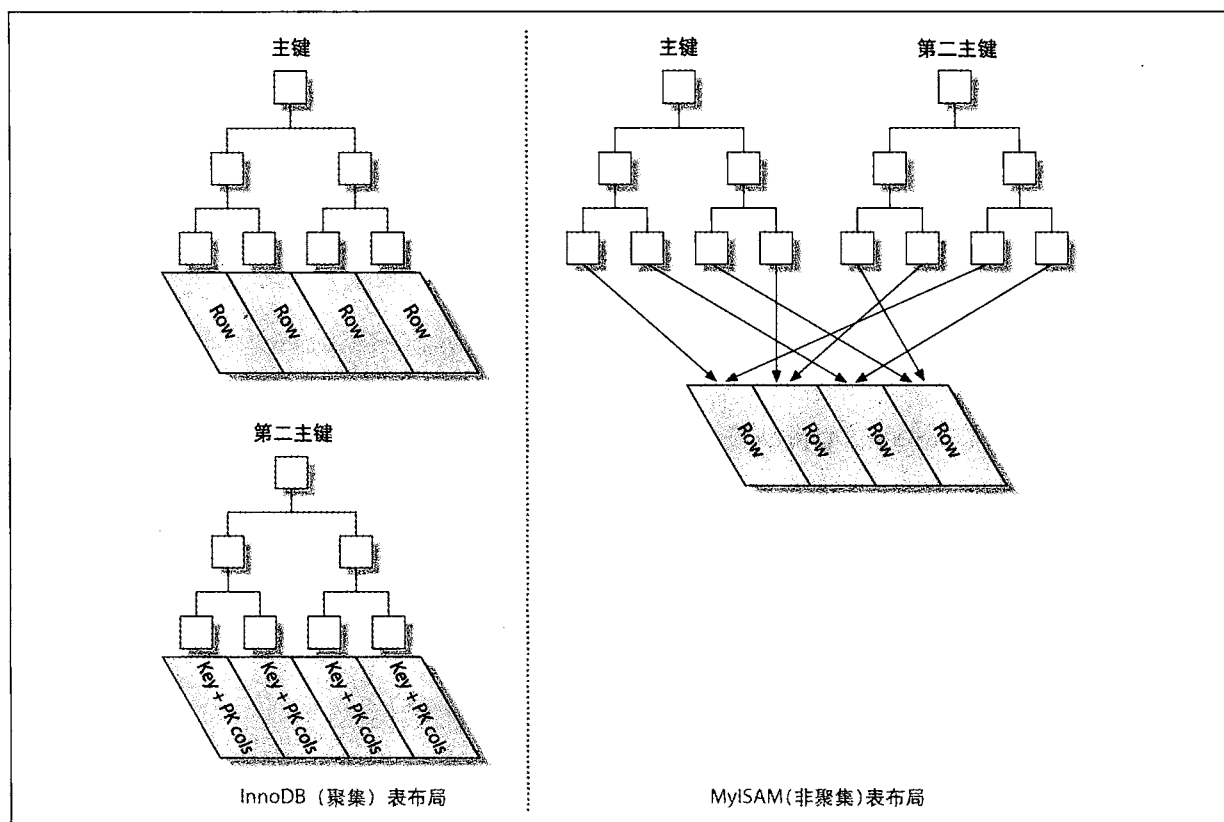


图 3-9：聚集和非聚集对比图

如果没有理解聚集和非聚集为何有区别、到底有何区别及区别的重要性，也不用担心。随着学习的内容越多，问题也会变得越清楚，尤其是在本章的后续部分及下一章。这些概念是复杂的，需要一些时间才能完全理解。

在 InnoDB 中按照主键顺序插入行

如果正在使用 InnoDB 并且不需要任何特定的聚集，就可以定义一个代理键 (Surrogate Key)。它是一种主键，但是值和应用程序无关。最简单的办法是使用 `AUTO_INCREMENT` 列。这会保证行是顺序插入的并且能提高使用主键联接的性能。

最好避免随机（乱序）聚集键。例如，使用 UUID 值是不好的选择：它使聚集索引插入是随机的，这是最坏的情形，并且会使数据聚集完全没有帮助。

为了进行展示，我们评测了两个例子。第 1 个是使用整数 ID 插入 `userinfo` 表，表定义如下：

```
CREATE TABLE userinfo (
  id          int unsigned NOT NULL AUTO_INCREMENT,
  name        varchar(64) NOT NULL DEFAULT '',
  email       varchar(64) NOT NULL DEFAULT '',
  password    varchar(64) NOT NULL DEFAULT '',
  dob         date DEFAULT NULL,
  address     varchar(255) NOT NULL DEFAULT '',
  city        varchar(64) NOT NULL DEFAULT ''
```

```
state_id      tinyint unsigned NOT NULL DEFAULT '0',
zip           varchar(8) NOT NULL DEFAULT '',
country_id    smallint unsigned NOT NULL DEFAULT '0',
gender        ('M','F') NOT NULL DEFAULT 'M',
account_type  varchar(32) NOT NULL DEFAULT '',
verified      tinyint NOT NULL DEFAULT '0',
allow_mail    tinyint unsigned NOT NULL DEFAULT '0',
parent_account int unsigned NOT NULL DEFAULT '0',
closest_airport varchar(3) NOT NULL DEFAULT '',
PRIMARY KEY (id),
UNIQUE KEY email (email),
KEY country_id (country_id),
KEY state_id (state_id),
KEY state_id_2 (state_id,city,address)
) ENGINE=InnoDB
```

注意到有个自增的整数主键。

第 2 个例子是 userinfo_uuid 表。它和 userinfo 表一样，但是主键是 UUID，而不是整数。

118

```
CREATE TABLE userinfo_uuid (
  uuid varchar(36) NOT NULL,
  ...
```

我们测试了这两个表。首先，我们在一个有足够内存容纳索引的服务器上向两个表插入 100 万条数据。接下来，我们再次插入 300 万行数据，使索引的大小超过服务器的内存容量。表 3-2 显示了测试结果。

表 3-2：向 InnoDB 表中插入数据的测试结果

| 表 | 行数 | 时间 (秒) | 索引大小 (MB) |
|---------------|-----------|--------|-----------|
| userinfo | 1 000 000 | 137 | 342 |
| userinfo_uuid | 1 000 000 | 180 | 544 |
| userinfo | 3 000 000 | 1233 | 1036 |
| userinfo_uuid | 3 000 000 | 4525 | 1707 |

注意到插入 UUID 主键行不仅花费的时间长，而且索引也会更大。一部分是因为主键更大，但是毫无疑问有一部分是因为分页和碎片。

为了明白为什么会发生这种情况，来看看向第 1 个表中插入数据会发生什么情况。图 3-10 显示了插满一个页面后就会开始第 2 个页面。

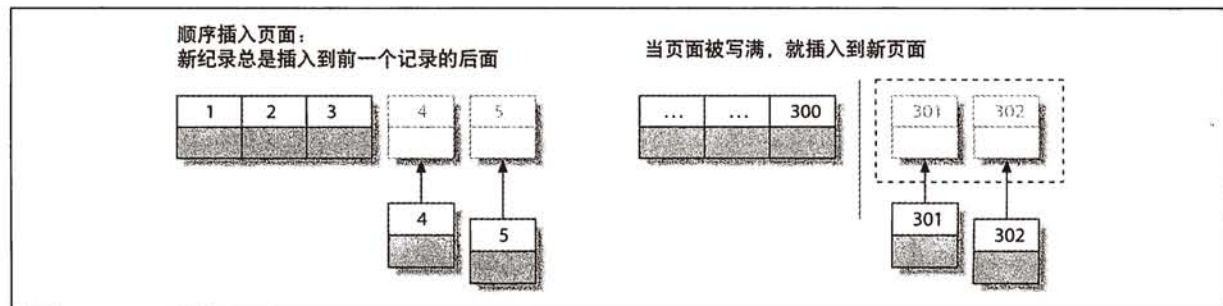


图 3-10：向聚集索引插入顺序索引值

如图 3-10 所示，InnoDB 把每条记录都保存在前一条记录的后面，因为主键值是顺序的。当达到页面最大的填

充因子 (Fill Factor) (InnoDB 初始的填充因子是 15/16, 为以后修改留出空间) 时, 第 2 条记录就会写入新页面。一旦数据按照这种顺序方式进行装载, 页面就会很紧凑, 近似于用顺序数据全部填满, 它就是想要的结果。

图 3-11 显示了使用 UUID 作为聚集索引进行插入, 我们看看会发生什么。

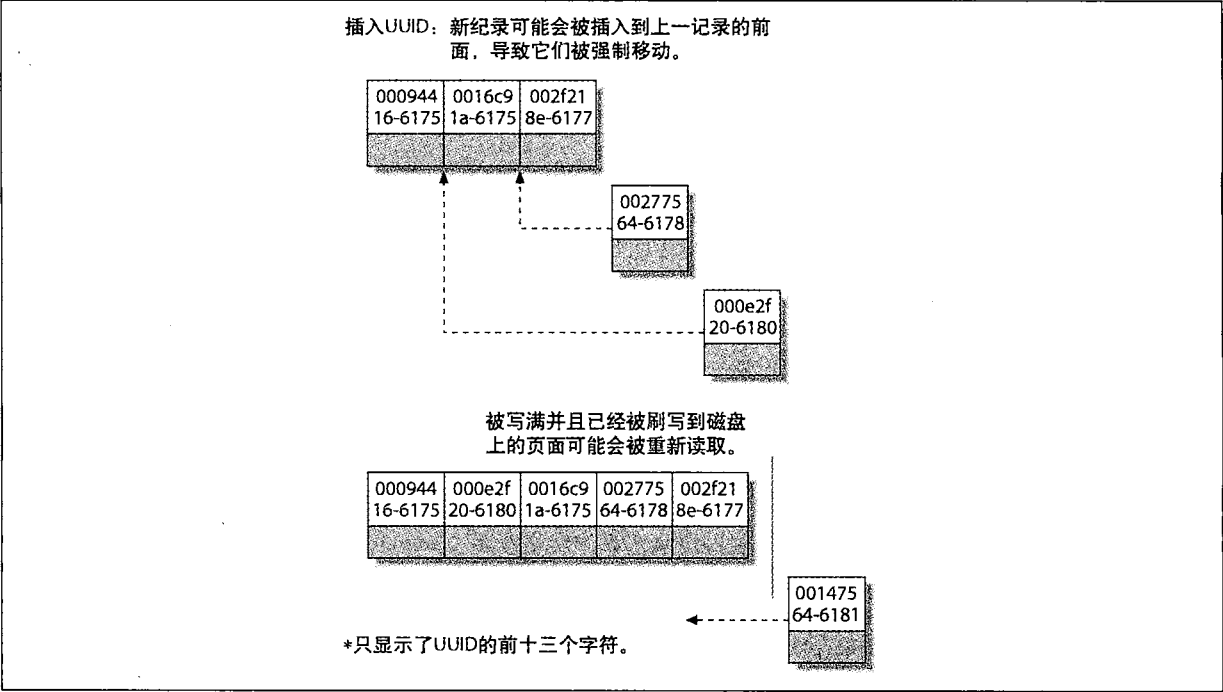


图 3-11: 向聚集索引中插入非顺序数据

由于新行的主键不一定比前一个大, 因此 InnoDB 不能总是把新行插入到索引的最后。它不得为新行寻找合适的位置——通常在已有数据的中段——并且为它分配空间。这会导致 大量的额外工作并且导致不优化的数据布局。下面是对缺点的总结:

- 目标页面也许会被刷写到磁盘上并且从缓存中移走, 无论哪种情况, InnoDB 都不得不在插入新行之前从磁盘上找到并读取它。这导致了大量的随机 I/O。
- InnoDB 有时不得不进行分页, 为新行开辟空间。这会导致移动大量数据。
- 页面会因为分页而变得稀疏和不规则地被填充, 因此最终的数据会有碎片。

在把这些随机数据装载到聚集索引之后, 你也许要使用 OPTIMIZE TABLE 来重建表并且优化填充页面。

这个例子的价值就在于使用 InnoDB 的时候要以主键的顺序插入数据, 并且要使用单调增的主键来插入新行。

主键顺序何时会造成更坏的结果

对于高并发工作负载, 根据 InnoDB 当前的实现, 按照主键顺序进行插入实际上会造成 InnoDB 内部单点竞争。这个“热点”是主键的上界。因为所有的插入都发生在那儿, 并发插入可能会竞争下一个键锁并且/或者 AUTO_INCREMENT 锁 (一个或两者都会是热点)。如果遇到这个问题, 你也许能重新设计表或应用程序, 或者对 InnoDB 进行调优, 以更好地执行这种负载。关于 InnoDB 调优的更多内容请参考第 6 章。

3.3.4 覆盖索引

Covering Indexes

索引是找到行的高效方式，但是 MySQL 也能使用索引来接收列的数据，这样就可以不用读取行数据。毕竟，索引的叶子节点包含了它们索引的数据，既然索引能给你需要的数据，为什么还要读取行呢？包含（或者“覆盖”）所有满足查询需要的数据的索引叫覆盖索引（Covering Index）。

覆盖索引是很有力的工具，可以极大地提高性能。考虑一下只需要读取索引，而不是数据的优势：

- 索引记录通常远小于全行大小，因此，如果只读取索引，MySQL 就能极大地减少数据访问量。这对缓存的负载是非常重要的，它大部分的响应时间都花在拷贝数据上。对于 I/O 密集型的负载也有帮助，因为索引比数据小很多，能更好地被装入内存。（这对于 MyISAM 尤其正确，它能压缩索引，使它们变得更小。）
- 索引是按照索引值进行排序的（至少在页内是如此），因此 I/O 密集型范围访问将会比随机地从磁盘上提取每一行数据要快得多。对于某些存储引擎，例如 MyISAM，甚至可以对表使用 OPTIMIZE 命令得到全部排序的索引，它能让简单的范围查询使用完全的顺序索引访问。
- 大部分存储引擎缓存索引比缓存数据更好。（Falcon 是一个著名的例外。）一些存储引擎，例如 MyISAM，只在 MySQL 的内存中缓存了索引。因为操作系统缓存了给 MyISAM 的数据，访问它通常需要系统调用。这也许会导致巨大的性能影响，尤其对于那些系统调用占了数据访问中最大开销的负载。
- 覆盖索引对于 InnoDB 表特别有用，因为 InnoDB 的聚集缓存。InnoDB 的第二索引在叶子节点中保存了行的主键值。因此，覆盖了查询的第二索引在主键中避免了另外一次索引查找。

121 在所有这些场景中，从索引中满足查询要求的代价一般要比查找行小得多。

覆盖索引和任何一种索引都不一样。覆盖索引必须保存它包含的列的数据。哈希、空间和全文索引不会保存这些值，因此 MySQL 只能使用 B-Tree 索引来覆盖查询。并且，不同的存储引擎实现覆盖索引的方式不一样，不是所有的存储引擎都支持覆盖索引（在写作本书的时候，Memory 和 Falcon 存储引擎不支持它）。

当发起一个被索引覆盖的查询（索引覆盖查询（Index-covered Query））时，会在解释器（EXPLAIN）的 Extra 列看到“使用索引（Using Index）”（注 12）。例如，sakila.inventory 表在（store_id, film_id）上有多列索引。MySQL 也能为只访问了这两列的查询使用索引，如下：

```
mysql> EXPLAIN SELECT store_id, film_id FROM sakila.inventory\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: inventory
         type: index
possible_keys: NULL
         key: idx_store_id_film_id
      key_len: 3
         ref: NULL
        rows: 4673
     Extra: Using index
```

索引覆盖查询能禁用掉这种优化。MySQL 查询优化器在执行查询之前会决定是否有索引覆盖它。假设索引覆盖了 WHERE 条件，但是不是整个查询。如果条件为假（False），MySQL 5.1 和之前版本无论如何都会提取这一行，

注 12：很容易把 Extra 列的“使用索引（Using Index）”和 type 列的“索引（index）”弄混淆。然而，它们完全不一样。type 列和覆盖索引没有任何关系，它显示了查询的访问类型，或者说是查询查找数据行的类型。

尽管它不需要并且会过滤掉这行数据。

让我们看看原因，以及如何重写查询以解决该问题。从下面的查询开始：

```
mysql> EXPLAIN SELECT * FROM products WHERE actor='SEAN CARREY'
-> AND title like '%APOLLO%'\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: products
         type: ref
possible_keys: ACTOR, IX_PROD_ACTOR
         key: ACTOR
      key_len: 52
         ref: const
        rows: 10
      Extra: Using where
```

122

该索引不能覆盖查询，有两个原因：

- 没有索引覆盖查询，因为从表中选择了所有的列，并且没有索引覆盖所有列。MySQL 理论上有一个捷径可以使用，但是，WHERE 子句只提到了索引覆盖的列，因此 MySQL 可以使用索引找到 actor 并检查 title 是否匹配，这只能通过读取整行进行。
- MySQL 不能在索引中执行 LIKE 操作。这是低层次存储引擎 API 的限制，它只允许在索引中进行简单比较。MySQL 能在索引中执行前缀匹配的 LIKE 模式是因为能把它们转化为简单比较，但是查询中前导的通配符是存储引擎无法转化匹配的。因此，MySQL 服务器自己将不得不提取和匹配行的数据，而不是索引值。

有办法可以解决这个问题，那就是合并索引及重写查询。可以把索引进行延伸，让它覆盖 (artist, title, prod_id) 并且按照下面的方式重写查询：

```
mysql> EXPLAIN SELECT *
-> FROM products
-> JOIN (
->   SELECT prod_id
->   FROM products
->   WHERE actor='SEAN CARREY' AND title LIKE '%APOLLO%'
-> ) AS t1 ON (t1.prod_id=products.prod_id)\G
***** 1. row *****
      id: 1
  select_type: PRIMARY
        table: <derived2>
      ...omitted...
***** 2. row *****
      id: 1
  select_type: PRIMARY
        table: products
      ...omitted...
***** 3. row *****
      id: 2
  select_type: DERIVED
        table: products
         type: ref
possible_keys: ACTOR, ACTOR_2, IX_PROD_ACTOR
         key: ACTOR_2
      key_len: 52
         ref:
        rows: 11
      Extra: Using where; Using index
```

现在，当 MySQL 在 FROM 子句的子查询中找到匹配行的时候，它会在查询的第一阶段使用覆盖索引。它没有使用索引来覆盖整个查询，但是这比什么都没有好。

这种优化的效率取决于 WHERE 子句中发现的行的数量。假设 products 表含有 100 万行。让我们看看这两个查询在 3 个不同的数据库上的性能，每个数据库都有 100 万行数据。

- 1. 在第 1 个数据库中，Sean Carrey 出演了 30 000 部作品，其中 20 000 部的标题中包含了 Apollo。
- 2. 在第 2 个数据库中，Sean Carrey 出演了 30 000 部作品，其中 40 部的标题中包含了 Apollo。
- 3. 在第 3 个数据库中，Sean Carrey 出演了 50 部作品，其中 10 部的标题中包含了 Apollo。

使用这 3 个数据库运行两个不同的查询，得到的结果如表 3-3 所示。

表 3-3：索引覆盖查询和非索引覆盖查询结果对比

| 数据集 | 原始查询 | 优化后的查询 |
|----------|--------------|--------------|
| Example1 | 每秒 5 个查询 | 每秒 5 个查询 |
| Example2 | 每秒 7 个查询 | 每秒 35 个查询 |
| Example3 | 每秒 2 400 个查询 | 每秒 2 000 个查询 |

下面是对结果的解释：

- 在 Example1 中，查询返回了一个巨大的结果集，因此不能看到优化的效果。大部分时间都花在了读取和发送数据上。
- 在 Exmpale2 中，经过索引过滤之后第 2 个条件只会返回少量的结果，这显示了优化的效果：性能提高了 5 倍。效率来自只须要读取 40 行，而不是第 1 个查询中的 30000 行。
- Example3 显示了子查询效率不高的情况。索引过滤后的结果集不大，所以子查询的代价就比读取表中的数据要高得多。

这种优化有时是帮助 MySQL 5.1 和以前版本避免读取不需要的行的有效方式。MySQL 6.0 自身有可能会避免额外的工作，因此在升级的时候也许能简化查询。

在大多数存储引擎中，索引能覆盖索引中的部分列。然而，InnoDB 实际上把这种优化更进一步。想想 InnoDB 在叶子节点中保存了主键值的第二索引。这意味着 InnoDB 的第二索引有效地拥有了 InnoDB 可以用来覆盖查询的“额外列（Extra Column）”。

例如，sakila.actor 表使用了 InnoDB 并且在 last_name 上有索引，因此，即使该列不是索引的一部分，索引也可以覆盖取得主键 actor_id 的查询：

```
mysql> EXPLAIN SELECT actor_id, last_name
-> FROM sakila.actor WHERE last_name = 'HOPPER'\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: actor
         type: ref
possible_keys: idx_actor_last_name
         key: idx_actor_last_name
      key_len: 137
         ref: const
         rows: 2
      Extra: Using where; Using index
```


3.3.5 为排序使用索引扫描

Using Index Scans for Sorts

MySQL 有两种产生排序结果的方式：使用文件排序 (Filesort)，或者扫描有序的索引（注 13）。EXPLAIN 的输出中 type 列的值为“索引 (Index)”，这说明 MySQL 会扫描索引。不要把这个和 Extra 列中的“使用索引 (Using Index)”混淆起来。

扫描索引本身是很快的，因为它只需要从一条索引记录移到另外一条记录。然而，如果 MySQL 没有使用索引覆盖查询，就不得不查找在索引中发现的每一行。这基本是随机 I/O 的，因此以索引顺序读取数据通常比顺序扫描表慢得多，尤其对于 I/O 密集的工作负载。

MySQL 能为排序和查找行使用同样的索引。如果可能，按照这样一举两得的方式设计索引是个好主意。

按照索引对结果进行排序，只有当索引的顺序和 ORDER BY 子句中的顺序完全一致，并且所有列排序的方向（升序或降序）一样才可以。如果查询联接了多个表，只有在 ORDER BY 子句的所有列引用的是第一个表才可以。查找查询中的 ORDER BY 子句也有同样的局限：它要使用索引的最左前缀。在其他所有情况下，MySQL 使用文件排序。

ORDER BY 无须定义索引的最左前缀的一种情况是前导列为常量。如果 WHERE 子句和 JOIN 子句为这些列定义了常量，它们就能弥补索引的缺陷。

例如，sakila 样例数据库中的 rental 表在 (rental_date、inventory_id、customer_id) 上面有索引：

```
CREATE TABLE rental (
...
PRIMARY KEY (rental_id),
UNIQUE KEY rental_date (rental_date,inventory_id,customer_id),
KEY idx_fk_inventory_id (inventory_id),
KEY idx_fk_customer_id (customer_id),
KEY idx_fk_staff_id (staff_id),
...
);
```

MySQL 使用 rental_date 索引对下面的查询进行排序，在解释器中不会看到 filesort：

```
mysql> EXPLAIN SELECT rental_id, staff_id FROM sakila.rental
-> WHERE rental_date = '2005-05-25'
-> ORDER BY inventory_id, customer_id\G
***** 1. row *****
      type: ref
possible_keys: rental_date
      key: rental_date
      rows: 1
      Extra: Using where
```

即使 ORDER BY 子句自身不是索引的最左前缀，它也能工作，因为它为索引的第 1 列定义了等于条件。

下面有更多可以使用索引进行排序的查询。该查询能工作的原因是查询为索引的第一列提供了一个常量，并且 ORDER BY 使用了第 2 列。它们合在一起形成了最左前缀：

```
... WHERE rental_date = '2005-05-25' ORDER BY inventory_id DESC;
```

注 13: MySQL 有两种排序算法，更多细节可以参考 176 页“排序优化”。

下面的查询也没有问题，因为 ORDER BY 中的两列都是索引的最左前缀：

```
... WHERE rental_date > '2005-05-25' ORDER BY rental_date, inventory_id;
```

下面是一些不能使用索引进行排序的查询：

- 这个查询使用了两种不同的排序方向，但是索引列都是按照升序排列的：
... WHERE rental_date = '2005-05-25' ORDER BY inventory_id DESC, customer_id ASC;
- 这儿，ORDER BY 引用了一个不在索引中的列：
... WHERE rental_date = '2005-05-25' ORDER BY inventory_id, staff_id;
- 这儿，WHERE 和 ORDER BY 不能形成索引的最左前缀：
... WHERE rental_date = '2005-05-25' ORDER BY customer_id;
- 这个查询在第 1 列上有范围条件，因此 MySQL 不会使用余下的索引：
... WHERE rental_date > '2005-05-25' ORDER BY inventory_id, customer_id;
- 这个查询在 inventory_id 列上有多个等于条件。对于排序来说，这也是范围查询：
... WHERE rental_date = '2005-05-25' AND inventory_id IN(1,2) ORDER BY customer_id;
- 这里虽有一个 MySQL 理论上可以使用索引对联接进行排序的例子，但实际却不成功，因为优化器把 film_actor 表放在联接的第二个位置上了（第 4 章展示了更改联接顺序的方式）：

```
mysql> EXPLAIN SELECT actor_id, title FROM sakila.film_actor
      -> INNER JOIN sakila.film USING(film_id) ORDER BY actor_id\G
+-----+-----+-----+
| table      | Extra |
+-----+-----+-----+
| film       | Using index; Using temporary; Using filesort |
| film_actor | Using index |
+-----+-----+-----+
```

按照索引进行排序的一个最重要的用途是有 ORDER BY 和 LIMIT 子句的查询。我们稍后会做更详细的研究。

3.3.6 压缩（前缀压缩）索引

Packed (Prefix-Compressed) indexes

MyISAM 使用前缀压缩（Prefix Compression）以减少索引大小、运行更多索引被装入内存，以及在某些情况下极大地提高性能。它在默认情况下会压缩字符串，但是可以让它压缩整数。

MyISAM 在对索引块排序的时候，首先对第 1 个值进行全排序，然后记录下有相同前缀的字节数，加上不同的值作为后缀。例如，如果第 1 个值是“perform”并且第 2 个值为“performance”，第 2 个值就会被近似地存储为“7,ance”。MyISAM 也能对相邻行指针使用前缀压缩。

压缩后的块占用的空间较小，但是使某些操作变慢了。因为每个值的压缩前缀依赖于前面的值，MyISAM 不能在索引块中使用二进制搜索找到想要的值，而必须从头开始。顺序向前的操作性能尚可，但是反向扫描——例如 ORDER BY DESC——不会很好地工作。任何须查找数据块中部的行的操作要对块进行扫描，平均说来，要扫描半个块。

测试表明压缩后的键使 MyISAM 表上的索引查找对 CPU 密集的负载会慢几倍，因为扫描须随机查找。反向扫

描甚至更慢。权衡压缩的标准是 CPU 内存资源和磁盘资源的折中。缩压后的索引大小也许只有原来的 1/10，并且，如果工作负载是 I/O 密集的，那么对于某些查询，它们就能更好地弥补性能。

可以在使用 CREATE TABLE 命令时用 PACK_KEYS 控制索引压缩的方式。

3.3.7 多余和重复索引

Redundant and Duplicate Indexes

MySQL 允许你在同一个列上创建多个索引，它不会注意到你的错误，也不会为错误提供保护。MySQL 不得不单独维护每一个索引，并且查询优化器在优化查询的时候会逐个考虑它们。这会严重地影响性能。

重复索引 (Duplicate Index) 是类型相同，以同样的顺序在同样的列上创建的索引。应该避免创建重复索引，并且在发现它的时候把它移除掉。

有时会在不经意间创建重复索引。例如，看看下面的代码：

```
CREATE TABLE test (
  ID INT NOT NULL PRIMARY KEY,
  UNIQUE(ID),
  INDEX(ID)
);
```

一个没有经验的用户也许会认为这表明该列的角色是主键，于是添加了一个 UNIQUE 约束，并且添加了一个查询使用的索引。事实上，MySQL 利用索引实现了 UNIQUE 约束和 PRIMARY KEY 约束，因此实际上这在同一列上创建了 3 个相同的索引！通常没有理由这么做，除非想在同一列上有不同的索引以满足不同类型的查询（注 14）。

多余索引 (Redundant Index) 和重复索引有一些不同。如果列 (A, B) 上有索引，那么另外一个列 (A) 上的索引就是多余的。这就是说，(A, B) 上的索引能被当成 (A) 上的索引。（这种多余只适合于 B-Tree 索引。）然而，(B, A) 上的索引不会是多余的，(B) 上的索引也不是，因为列 B 不是列 (A, B) 的最左前缀。还有，不同类型的索引（例如哈希或全文索引）对于 B-Tree 索引不是多余的，无论它们针对的是哪一列。

多余索引通常发生在向表添加索引的时候，例如，有人也许会在 (A, B) 上添加索引，而不是对 (A) 上已有的索引进行扩展，以覆盖 (A, B)。

在大部分情况下，多余索引都是不好的，为了避免它，应该扩展已有索引，而不是添加新索引。但是，还是有一些情况出于性能考虑需要多余索引。使用多余索引的主要原因是扩展已有索引的时候，它会变得很大。

例如，如果在一个整数列上有索引，并且把它扩展到一个很长的 VARCHAR 列，该索引就会变得很慢。当查询把索引当成覆盖索引，或者表是 MyISAM 表并且上面执行大量范围扫描（由于 MyISAM 的前缀压缩）的时候，尤为正确。

考虑一下 userinfo 表，我们已经在 117 页“在 InnoDB 中按照主键顺序插入行”描述过它，也在本章开头提到过。这个表有 1 000 000 行数据，并且每个 state_id 都有大约 20 000 条记录。在 state_id 列上有一个索引，它对于下面的查询是有用的。把下面的查询叫做 Q1：

```
mysql> SELECT count(*) FROM userinfo WHERE state_id=5;
```

注 14：如果索引类型不同，就无所谓重复了。通常都会有好的理由要求有 KEY(col) 和 FULLTEXT KEY(col)。

一个简单的测试表明执行效率大约为 115 个查询每秒 (Queries Per Second, QPS)。还有一个相关查询, 获得了几列的数据, 而不是只统计行数, 把它叫做 Q2:

```
mysql> SELECT state_id, city, address FROM userinfo WHERE state_id=5;
```

对于这个查询, 测试结果低于 10 QPS (注 15)。简单解决方案是把索引扩展到 (state_id, city, address) 以提高性能, 这样索引就能覆盖查询:

```
mysql> ALTER TABLE userinfo DROP KEY state_id,
-> ADD KEY state_id_2 (state_id, city, address);
```

在扩展索引之后, Q2 运行得更快了, 但是 Q1 变慢了很多。如果想让两个查询都变快, 就应该保留这两个索引, 即使单列索引是多余的。表 3-4 显示了这两个查询和索引策略的详细结果, 存储引擎分别是 MyISAM 和 InnoDB。注意到 InnoDB 上 Q1 的性能在只使用 state_id_2 索引的情况下没有下降很多, 因为 InnoDB 没有使用任何键压缩。

表 3-4: 使用不同索引策略的 SELECT 查询的 QPS 测试结果

| | 只有 state_id | 只有 stat_id_2 | 同时有 state_id 和 state_id_2 |
|------------|-------------|--------------|---------------------------|
| MyISAM, Q1 | 114.96 | 25.40 | 112.19 |
| MyISAM, Q2 | 9.97 | 16.34 | 16.37 |
| InnoDB, Q1 | 108.55 | 100.33 | 107.97 |
| InnoDB, Q2 | 12.12 | 28.04 | 28.06 |

有两个索引的缺点就是维护开销。表 3-5 显示了向表中插入 100 万条数据需要的时间。

表 3-5: 使用不同索引策略插入 100 万条数据的速度

| | 只有 state_id | 同时有 state_id 和 state_id_2 |
|----------------------|-------------|---------------------------|
| InnoDB, 两个索引都有足够的内存 | 88 秒 | 136 秒 |
| MyISAM, 只有一个索引有足够的内存 | 72 秒 | 470 秒 |

如你所见, 向有更多索引的表中插入新行慢得多。这通常是对的: 添加新索引可能会对 INSERT、UPDATE 和 DELETE 有较大的性能影响, 尤其在新索引遇到内存限制的时候。

3.3.8 索引和锁定

Indexing and Locking

索引对 InnoDB 有很重要的作用, 因为它会让查询锁定更少的列。这是重要的想法, 因为在 MySQL 5.0 中, InnoDB 只有在事务提交之后才会给行解锁。

如果查询永远不会访问不需要的行, 它们就会锁定更少的行, 有两个原因使它更有利于性能。首先, 尽管 InnoDB 的行锁定效率很高, 使用的内存很少, 但是锁定的时候还是有一些开销。其次, 锁定超过需要的行会增加锁竞争 (Lock Contention) 和减少并发。

注 15: 这儿使用的是内存中示例 (in-memory example)。当表更大并且工作负载变得 I/O 密集的时候, 数字的区别会更大。

InnoDB 只有在访问行的时候才锁定它们，并且索引能减少 InnoDB 访问的行数，从而减少锁定。然而，只有当 InnoDB 能在存储引擎级过滤掉不需要的行的时候才能起作用。如果索引不允许 InnoDB 那么做，MySQL 就不得不在 InnoDB 取得行并且把它们返回给服务器级之后再使用 WHERE 子句。这时，已经无法避免锁定行了：InnoDB 已经锁定了它们，并且服务器不能解锁。

以下例子更好地解释了这种情况，还是使用 Sakila 数据库：

```
mysql> SET AUTOCOMMIT=0;
mysql> BEGIN;
mysql> SELECT actor_id FROM sakila.actor WHERE actor_id < 5
-> AND actor_id <> 1 FOR UPDATE;
+-----+
| actor_id |
+-----+
| 2        |
| 3        |
| 4        |
+-----+
```

该查询只返回 2 到 4 行，但是它实际以独占的方式锁定了 1 到 4 行。InnoDB 锁定第 1 行的原因是，MySQL 为该查询选定的计划是索引范围访问：

```
mysql> EXPLAIN SELECT actor_id FROM sakila.actor
-> WHERE actor_id < 5 AND actor_id <> 1 FOR UPDATE;
+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | key | Extra |
+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | actor | range | PRIMARY | Using where; Using index |
+-----+-----+-----+-----+-----+-----+
```

换句话说，低层次的存储引擎操作是“从索引的开头开始，并且提取所有行直到 actor_id < 5 不成立。”服务器不会告诉 InnoDB 能消除第一行的 WHERE 条件。注意到解释器的 Extra 列中出现了“使用 Where (Using Where)”。这意味着 MySQL 在存储引擎返回行之后使用了 WHERE 过滤条件。

索引策略总结

现在你已经了解了更多关于索引的知识，但也许还不知道如何从自己的表开始。虽然最重要的事情是检查自己最常运行的查询，但是也应该考虑不那么经常进行的操作，比如插入和更新数据。要避免在不知道什么查询会使用索引之前就创建它这种常见的错误，并且要考虑自己是否所有的索引能形成一个优化的配置。

有时只从查询就可以知道需要什么索引，只要把它们加上就可以了。但是有时有各种类型的查询，却不能找到适合它们的完美索引，这就需要一些折中。为了找到最佳平衡，应该进行测试和剖析。

第一个要检查的就是响应时间。要考虑为任何耗时很长的查询添加索引。然后检查导致最大负载的查询（更多衡量的手段请参阅第 2 章），并且添加索引支持它们。如果系统正好碰到了内存、CPU 或磁盘瓶颈，也要把它们考虑进去。例如，如果运行了很多很长的聚合查询以生成汇总，那么磁盘使用会因为支持 GROUP BY 查询的覆盖索引而得益。

在任何可能的地方，都要试着扩展索引，而不是新增索引。通常维护一个多列索引要比维护多个单列索引容易。如果不知道查询的分布，就要尽可能地使索引变得更有选择性，因为高选择性的索引通常更有好处。

下面的查询证明了 row1 被锁住，尽管它没有出现在第 1 个查询的结果中。让第一个连接保持开启状态，开始

第 2 个连接执行下面的查询：

131

```
mysql> SET AUTOCOMMIT=0;
mysql> BEGIN;
mysql> SELECT actor_id FROM sakila.actor WHERE actor_id = 1 FOR UPDATE;
```

这个查询将会停止，等待第 1 个查询释放 row1 上的锁。该行为对基于命令的复制（Statement-based Replication，在第 8 章中讨论）正常工作是必需的。

如同这个例子所显示的，即使 InnoDB 使用了索引，它也能锁定不需要的行。这个问题在它不能使用索引找到并锁定行的时候会更严重：如果没有索引，MySQL 不管是否需要行，都会进行全表扫描并且锁定每一行（注 16）。

这里有一个很少人知道的关于 InnoDB、索引和锁定的细节：InnoDB 能在第二索引上放置共享（读取）锁，但是独占（写入）锁要求访问主键。这消除了使用覆盖索引的可能性，并且能导致 SELECT FOR UPDATE 比 LOCK IN SHARE MODE 或非锁定查询慢得多。

3.4 索引实例研究

An Indexing Case Study

理解索引最容易的方式是结合例子，因此这里准备了一个索引的示例。

假设我们要设计一个带有用户资料的在线约会网站，用户资料有很多列，例如用户的 country、state/region、city、sex、age、eye color 等。站点必须支持使用这些条件的组合来搜索用户。它还必须对用户排序，并且按照用户上次在线时间、其他会员的评价等对结果进行限制。如何为这样一个复杂的需求设计索引？

奇怪的是，第一件事情是决定是否需要基于索引的排序，或者文件排序是否可以接受。基于索引排序限制了建立索引和查询的方式。例如，如果某个查询使用索引根据其他用户的评价对用户进行排序，就不能对 WHERE age BETWEEN 18 AND 25 这样的 WHERE 子句使用索引。如果 MySQL 对查询中的范围判据使用了索引，就不能对排序使用另外的索引（或者同一个索引的后缀）。假设这是很常见的 WHERE 子句，我们就会理所当然地认为许多查询会需要文件排序。

3.4.1 支持多种过滤条件

Supporting Many Kinds of Filtering

132

现在须检查哪个列有很多独特的值，以及哪个列在 WHERE 子句中出现得最频繁。拥有许多唯一值的列上的索引非常有选择性。这通常是好事情，因为它可让 MySQL 更有效地过滤掉不想要的列。

country 列可能有，也可能没有选择性，但是，它可能会在大多数查询中出现。sex 列肯定没有选择性，但是它有可能在每一个查询中存在。想到这一点，我们为许多不同的列组合创建了一系列索引，前缀是 (sex, country)。

习惯性的想法是不要在选择性很差的列上建索引。因此，为什么我们要在每个索引的开头放置一个无选择性的

注 16：曾经想过这会在拥有基于行的二进制日志和 READ COMMITTED 事务隔离层的 MySQL 5.1 中得到修复，但是它存在于我们测试的所有 MySQL 版本，包括 5.1.22。

列呢？我们有什么不对劲吗？

这么做有两个原因。第一个原因就像刚才说的那样，每个查询基本都会使用 `sex`，甚至会把站点设计为用户可以只用性别进行搜索。但是更重要的原因是，加上该列没什么坏处，因为我们有使用它的特殊诀窍。

诀窍就是：即使查询没有用性别来限制结果，也可以通过加上 `AND sex IN('m', 'f')` 来保证该索引会被用上。它不会过滤掉任何数据行，因此，结果就和没有在 `WHERE` 子句中包括 `sex` 一样。然而，我们须包括该列，因为它会让 MySQL 使用更大的索引前缀。这个诀窍对于类似的情形是有利的，但是，如果列有很多唯一的值，它就不会很好地工作，因为 `IN()` 列表太大了。

这个例子显示了一个通用的规则：保持表上的所有选项。当你设计索引的时候，不要只想着已有查询需要的索引，也要想着优化查询。如果看到需要某个索引，但是一些查询会因它而受到损害，就要问问自己是否应该改变这些查询。应该一起优化查询和索引，以找到最佳的折中。没有必要闭门造车，以得到最好的索引。

接下来，我们会考虑看到的 `WHERE` 条件的组合并且考虑哪个组合在没有正确索引的情况下会变得更慢。在 `(sex, country, age)` 上的索引很明显是一个选择，并且也许还会看到 `(sex, country, region, age)` 上的索引，以及 `(sex, country, region, city, age)` 上的索引。

这意味着有很多索引。如果想重新使用索引并且它不会产生太多的条件组合，我们可以使用 `IN()`，并且废弃掉 `(sex, country, age)` 和 `(sex, country, region, age)` 索引。如果它们没有在搜索中出现，则可以通过定义国家列表，或者国家区域列表来确保索引前缀拥有相等性约束。（所有国家、所有地区、所有性别的合并列表也许会太大了。）

这些索引会满足大部分常见的搜索查询，但是，如何为不怎么常见的选项，比如 `has_picture`、`eye_color`、`hair_color` 和 `education` 设计索引？如果这些列都非常没有选择性，并且不会被经常使用，我们可以简单地跳过它们并让 MySQL 多扫描几行。

替代的方法就是要把它们放在 `age` 列之前，并且使用前面介绍的 `IN()` 诀窍来处理没有定义它们的情况。

你也许已经注意到 `age` 列出现在索引的最后。什么使该列如此特殊呢？为什么它会在索引的最后？我们试着确保 MySQL 尽可能多地使用索引列，因为它只使用了最左前缀，并直到包括第一个使用了范围的条件。我们提到的所有其他列都可以在 `WHERE` 子句中使用相等性条件，但是 `age` 基本可以肯定是一个范围（比如，`age BETWEEN 18 AND 25`）。

我们可以把它转换到 `IN()` 列表，例如 `age IN(18, 19, 20, 21, 22, 23, 24, 25)`，但是这并不总能适合该类型的查询。这里要说的就是把范围判据放在索引的最后，这样优化器就会尽可能多地使用索引。

前面已经说过可以给索引添加越来越多的列，并且使用 `IN()` 列表来覆盖那些不在 `WHERE` 子句中的列，但是，如果过火了就会有麻烦。使用太多这种列表会造成优化器要计算的组合激增，并且会最终减慢查询速度。考虑下面的 `WHERE` 子句：

```
WHERE eye_color IN('brown','blue','hazel')
      AND hair_color IN('black','red','blonde','brown')
      AND sex IN('M','F')
```

优化器将会把这个转换为 $4 \times 3 \times 2 = 24$ 种组合，并且 `WHERE` 子句将不得不挨个检查它们。24 并不是很大的数字，但是数字上千就要注意了。老版本的 MySQL 在大量的 `IN()` 组合上有更多的问题：查询优化可能会比执行花费更多的时间并且消耗掉大量内存。较新的 MySQL 版本组合太多的时候会停止计算组合，这也限制了 MySQL 很好地使用索引。

3.4.2 避免多个范围条件

An MySQL Interview Problem

我们假设有一个 `last_online` 列并且希望通过下面的查询显示上周在线的用户：

```
WHERE eye_color IN('brown','blue','hazel')
      AND hair_color IN('black','red','blonde','brown')
      AND sex IN('M','F')
      AND last_online > DATE_SUB('2008-01-17', INTERVAL 7 DAY)
      AND age BETWEEN 18 AND 25
```

这个查询有一个问题：它有两个范围条件。MySQL 能为 `last_online` 或 `age` 使用索引，但是不能为两个都使用。

134

什么是范围条件

解释器 (EXPLAIN) 的输出有时候让人不易识别 MySQL 到底是真正地查找了范围值，还是一系列值。解释器使用了同样的词：“range”来描述这两种情况。例如，MySQL 调用了下面的“范围”查询，如同 `type` 列显示的：

```
mysql> EXPLAIN SELECT actor_id FROM sakila.actor
-> WHERE actor_id > 45\G
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: actor
      type: range
```

但是这个又如何？

```
mysql> EXPLAIN SELECT actor_id FROM sakila.actor
-> WHERE actor_id IN(1, 4, 99)\G
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: actor
      type: range
```

通过看解释器无法分辨出差别，但是我们可以从值的范围和多个查询相等条件得出不同。我们认为第 2 个查询就是多个相等条件。

我们并不是挑剔：这两种索引访问性能不同。范围条件使 MySQL 忽略了索引中的其他列，但是多个相等条件不会有这个局限。

如果 `last_online` 限制在没有 `age` 限制的时候出现，或者 `last_online` 比 `age` 更有选择性，可能就会想在索引的最后加上 `last_online`。但是，如果不能把 `age` 转换为 `IN()` 列表，又想同时突破 `last_online` 和 `age` 的限制，该怎么办？此时没有直接的解决办法，但是可以把一个范围查询转换为相等性比较。为了做到这点，我们添加了一个预先计算好的 `active` 列，并用一个周期性任务维护它。我们会在用户登录的时候把该值设置为 1，并且任务会在用户连续 5 天没有登录之后把该值设置为 0。

这个方法可让 MySQL 使用诸如 (`active`, `sex`, `country`, `age`) 这样的索引。列也许不是绝对精确，但是此种查询也不会要求很高的精确度。如果需要精确性，可以把 `last_online` 条件留在 `WHERE` 子句中，但是不索引

它。这种技巧和本章前面用来模拟 URL 查找的 HASH 索引有点类似。这个条件不会使用任何索引，但是因为索引会发现自己没有很大的帮助，而丢掉很多能找到的行。换句话说，缺乏索引也不会明显损伤查询。

现在，你也许已经了解了这种模式：如果用户想看到活动和不活动的用户，可以添加 IN() 列表。我们已经添加了大量的列表，一个替代措施就是创建不同的索引，以便满足要过滤的每种列组合。至少需要下面的索引：(active, sex, country, age)、(active, country, age)、(sex, country, age) 及 (country, age)。尽管这些索引对于单个特定的查询会更优化，但是维护它们的开销、它们所需要的空间可能会使这个整体变成不好的索引策略。

这是优化器更改可以真正影响优化的索引策略的例子。如果 MySQL 的将来版本可以做到松散索引扫描，它就能在单个索引上使用多个范围条件，这样就不再需要这里的 IN() 列表了。

3.4.3 优化排序

Denormalizing Sorting

该例子要讨论的最后一个问题是排序。文件排序对少量结果排序是挺快的，但是，如果是上百万行会怎样？例如，如果 WHERE 子句只定义了性别会如何？

可以为低选择性列排序添加一个特殊的索引。例如，(sex, rating) 列上的索引可以用于下面的查询：

```
mysql> SELECT <cols> FROM profiles WHERE sex='M' ORDER BY rating LIMIT 10;
```

这个查询有 ORDER BY 和 LIMIT 子句，并且在没有索引的时候会很慢。

即使有了索引，如果用户界面是分页的并且一些用户请求的不是靠近开头的页面，这个查询也会很慢。下面的例子利用偏移创建了一种很坏的 ORDER BY 和 LIMIT 组合：

```
mysql> SELECT <cols> FROM profiles WHERE sex='M' ORDER BY rating LIMIT 100000, 10;
```

这个查询无论有没有被索引都会是严重的问题，因为高偏移量要求它们花费很多时间来扫描将被丢掉的数据。非规范化 (Denormalizing)、预先计算 (Precomputing) 或缓存 (Caching) 有可能是解决该类查询的唯一办法。一个更好的策略是限制让用户查看的页数。这不可能影响用户的体验，因为没有人会真正在意搜索结果的第 10 000 页。

另外一种优化这种查询的好策略就是只提取最终需要的行的主键列。然后把它再联接回去以取得所有需要的列。这有助于最小化 MySQL 必须进行的收集最终会丢掉的数据的工作。下面是一个在 (sex, rating) 上需要索引，以高效工作的例子：

```
mysql> SELECT <cols> FROM profiles INNER JOIN (
->     SELECT <primary key cols> FROM profiles
->     WHERE x.sex='M' ORDER BY rating LIMIT 100000, 10
-> ) AS x USING(<primary key cols>);
```

3.5 索引和表维护

Indexing and Table Maintenance

即使已经用正确的数据类型创建了表并且添加了索引，工作也没有结束：还须维护表和索引以确保它们能很好地工作。表维护的 3 个主要目标是查找和修复损坏、维护精确的索引统计，并且减少碎片。

3.5.1 查找并修复表损坏

Handling Database File Corruption

表发生的最坏的事情是损坏 (Corruption)。在 MyISAM 中，通常是由于崩溃导致的。然而，所有存储引擎都会经历硬件问题、内部缺陷或操作系统导致的索引损坏。

损坏的索引能导致查询返回不正确的结果，在没有重复值的时候提高重复键的错误，或者甚至导致死锁和崩溃。如果遇到了古怪的行为——例如你认为不该发生的错误——就运行 `CHECK TABLE` 以确定表是否损坏了。(注意一些存储引擎不支持这个命令，其他支持多个选项来定义完全检查整个表的方式。) `CHECK TABLE` 通常能抓到大部分表和索引错误。

可以使用 `REPAIR TABLE` 命令来修复损坏的表，但是再次说明，不是所有的存储引擎都支持。在有些情况下，可以执行“无操作 (No-op)” `ALTER`，例如更改一个表，让它使用当前的存储引擎。下面是一个针对 InnoDB 表的例子：

```
mysql> ALTER TABLE innodb_tbl ENGINE=INNODB;
```

或者可以使用离线的引擎相关的修复工具，比如 `myisamchk`，或者转储数据并且重新加载它。然而，如果损坏存在于系统区域，或者在表的“行数据 (Row Data)”区域，而不是索引，就可能不能使用这些选择。在这种情况下，你也许要从备份中恢复表，或者尝试从损坏的文件中恢复数据 (参阅第 11 章)。

3.5.2 更新索引统计

Updating Index Statistics

MySQL 查询优化器在决定如何使用索引的时候会调用两个 API，以了解索引如何分布。第 1 个是调用 `records_in_range()`，它接受范围结束点并且返回 (也许是估计的) 该范围内记录的数量。第 2 个是 `info()`，它返回不同类型的数据，包括数据基数性 (每个键值有多少记录)。

137

当存储引擎没有向优化器提供查询检查的行的精确数量的时候，优化器会使用索引统计来估计行的数量，统计可以通过运行 `ANALYZE TABLE` 重新生成。MySQL 的优化器基于开销，并且主要的开销指标是查询会访问多少数据。如果统计永远没有产生，或者过时了，优化器就会做出不好的决定。解决方案是运行 `ANALYZE TABLE`。

每个存储引擎实现索引统计的方式不同，由于运行 `ANALYZE TABLE` 的开销不同，所以运行它的频率也会不一样：

- Memory 存储引擎根本就不保存索引统计。
- MyISAM 把索引统计保存在磁盘上，并且 `ANALYZE TABLE` 执行完整的索引扫描以计算基数性。整个表都会在这个过程中被锁住。
- InnoDB 不会把统计信息保存到磁盘上，但不是使用随机索引估计它们，而是在第一次打开表的时候利用随机索引进行估计。InnoDB 上的 `ANALYZE TABLE` 命令就使用了随机索引，因此 InnoDB 统计不够精确，除非让服务器运行很长的时间，否则不要手动更新它们。同样，`ANALYZE TABLE` 在 InnoDB 不是阻塞性的，并且相对不那么昂贵，因此可以在不大影响服务器的情况下在线更新统计。

可以使用 `SHOW INDEX FROM` 命令检查索引的基数性。例如：

```
mysql> SHOW INDEX FROM sakila.actor\G
***** 1. row *****
      Table: actor
    Non_unique: 0
```

```

    Key_name: PRIMARY
Seq_in_index: 1
Column_name: actor_id
Collation: A
Cardinality: 200
Sub_part: NULL
Packed: NULL
Null:
Index_type: BTREE
Comment:
***** 2. row *****
    Table: actor
Non_unique: 1
Key_name: idx_actor_last_name
Seq_in_index: 1
Column_name: last_name
Collation: A
Cardinality: 200
Sub_part: NULL
Packed: NULL
Null:
Index_type: BTREE
Comment:

```

该命令给出了大量的索引信息，MySQL 手册详细地解释了它们。但是我们想要你注意 Cardinality 列。它显示了存储引擎估计的索引中唯一值的数量。在 MySQL 5.0 及以上版本中可以通过 INFORMATION_SCHEMA.STATISTICS 表得到这些数据，它也非常方便。例如，可以对 INFORMATION_SCHEMA 表写查询来查找选择性很低的索引。

3.5.3 减少索引和数据碎片

Reducing Index and Data Fragmentation

B-Tree 索引能变成碎片，它降低了性能。碎片化的索引可能会以很差或非顺序的方式保存在磁盘上。

根据设计，B-Tree 索引须要随机磁盘访问以“下潜 (Dive)”到叶子页面上，因此随机访问是一种原则，而不是例外。然而，如果叶子页面顺序地并且紧密地排在一起，它的性能就会好些。如果不是，就认为它碎片化了，并且范围扫描或全索引扫描会慢很多倍。这对于索引覆盖的查询尤其正确。

表的数据存储也能变得碎片化。然而，数据存储碎片比索引碎片复杂得多。有两种类型的数据碎片：

行碎片 (Row fragmentation)

当行被存储在多个地方的多个片段中时，就会是这种碎片。即使查询只从索引中找一行数据，行碎片也会降低性能。

内部行碎片 (Intra-row fragmentation)

当逻辑上顺序的页面或行在磁盘上没有被顺序存储的时候，就会产生这种碎片。它影响了诸如全表扫描和聚集索引范围扫描这样的操作。这些操作通常从磁盘上的顺序数据布局得益。

MyISAM 表也许会遇到这两种碎片，但是 InnoDB 不会在短行上产生碎片。

为了消除数据碎片，可以运行 OPTIMIZE TABLE 或转储并重新加载数据。

这些操作对于大部分存储引擎都有用。对一些存储引擎，比如 MyISAM，它们也能通过排序算法重建索引来消除索引碎片。限制没有办法消除 InnoDB 索引的碎片，因为 MySQL5.0 中的 InnoDB 不能通过排序建立索引（注 17）。即使删掉并重新创建索引，也可能导致索引碎片，这取决于数据。

对于不支持 OPTIMIZE TABLE 的存储引擎，可以使用无操作 ALTER TABLE 重新建立表。如果是更改表，可让它使用当前的存储引擎：

```
mysql> ALTER TABLE <table> ENGINE=<engine>;
```

3.6 正则化和非正则化

Normalization and Denormalization

通常有很多方式表示给定的数据，全部正则化（Normalized）、全部非正则化（Denormalized）及中间方式。在正则化数据库中，每个因素只会被表达一次。相反，在非正则化数据库中，信息是重复的，或者保存在多个地方。

如果不熟悉正则化，就应该研究它。有很多关于它的好书籍和在线资料。这里我们只简单介绍跟本章相关的面。让我们从经典的 employees、departments 和 department heads 开始。

| EMPLOYEE | DEPARTMENT | HEAD |
|----------|-------------|-------|
| Jones | Accounting | Jones |
| Smith | Engineering | Smith |
| Brown | Accounting | Jones |
| Green | Engineering | Smith |

这个架构的问题在于数据更改的时候会出现异常。以 Brown 成为 Accounting 部门的 head 为例，须要更新多列以反映变化，并且那些更新会使数据处于不连续的状态。如果“Jones”行的 head 和“Brown”行的 head 不相同，就不知道哪个是正确的。就像老话说的那样，“戴两块表的人不知道正确的时间。”还有，我们不能表示一个没有 employee 的 department，如果删除 Accounting 部门的所有员工，就丢失 department 自身的信息。为了避免这个问题，需要把 employee 实体和 department 实体分开，以正则化该表。这个过程会导致下面的两个表：

| EMPLOYEE_NAME | DEPARTMENT |
|---------------|-------------|
| Jones | Accounting |
| Smith | Engineering |
| Brown | Accounting |
| Green | Engineering |

部门表：

| DEPARTMENT | HEAD |
|-------------|-------|
| Accounting | Jones |
| Engineering | Smith |

注 17：写作本书的时候，InnoDB 的开发人员正在解决这个问题。

这些表都采用了第二范式 (Second Normal Form)，它对于很多用途都足够了。然而，第二范式仅仅是多种范式中的一种。



提示：出于演示目的，这里使用姓作为主键，因为它是数据的“自然的标识符”。然而，在实际情况中，我们不会这么做。它不能保证唯一性，并且通常使用长字符串作主键都是不可取的。

3.6.1 正则化架构的利弊

Pros and Cons of a Normalized Schema

人们在寻求性能方面的帮助时，通常被建议正则化架构，尤其当负载写入密集型的时候。这通常都是好的建议，原因如下：

- 正则化更新通常比非正则化更新快。
- 当数据被很好地正则化之后，就很少，甚至没有重复数据，因此改动的数据会变少。
- 正则化表通常较小，因此更容易被装入内存并且性能更好。
- 由于缺少冗余数据，那么在取得数据的时候会较少采用 DISTINCT 或 GROUP BY。以前面的例子为例，如果不使用 DISTINCT 或 GROUP BY，就无法从非正则化架构中取得唯一的 department 列表，但是，如果 DEPARTMENT 是一个单独的表，那么就是很简单的查询。

正则化架构通常在抓取数据方面有缺点。任何一个在正则化架构上的非一般性查询 (Nontrivial Query) 都至少需要一个联接，或者多个。这不仅昂贵，而且会使某些索引策略变得不可能。例如，正则化会把某些从同一个索引中得到好处的列放到不同的表里面。

3.6.2 非正则化架构的利弊

Pros and Cons of a Denormalized Schema

非正则化架构由于所有数据都在一个表里面，避免了联接，所以性能不错。

如果不需要联接表，对于大多数查询，最坏的情况——甚至不使用索引的查询——是全表扫描。在数据不能被装入数据库的时候，这会比联接快得多，因为它避免了随机 I/O。

单个表也能运行更有效率的索引策略。假设有一个网站，用户在上面发布信息，并且一些用户是高级用户。现在你想查看每个高级用户发表的最新的 10 条消息。如果已经正则化了架构并且索引了消息发布的时间，查询就会像这样：

```
mysql> SELECT message_text, user_name
-> FROM message
-> INNER JOIN user ON message.user_id=user.id
-> WHERE user.account_type='premium'
-> ORDER BY message.published DESC LIMIT 10;
```

为了有效执行该查询，MySQL 须扫描 message 表上的 published 索引。对于找到的每一行，它都要探测 user 表并且检查用户是否为高级用户。如果只有一小部分是高级用户，就会显得效率低下。

另外一个可能的查询是从 user 表开始，选择所有的高级用户，取得他们的所有消息，然后执行文件排序。这

可能会更差。

问题在于联接，它不能让你同时使用一个索引进行排序和过滤。如果通过合并表并且在 (account_type, published) 上添加索引非正则化了数据，就可以不使用联接来写这个查询，它效率更高：

```
mysql> SELECT message_text, user_name
-> FROM user_messages
-> WHERE account_type='premium'
-> ORDER BY published DESC
-> LIMIT 10;
```

3.6.3 结合正则化和非正则化

正则化和非正则化都有自己的优缺点，那么如何选择最优方案呢？

真实情况是全部正则化或全部非正则化都过于学术性了，它们通常无助于真实情况。在真实世界中，通常要结合这两种方案，也许是采用部分正则化架构、缓存表及其他技巧。

非正则化数据最常见的技巧是复制、缓存、把一个表中的数据选到另一个表中。在 MySQL 5.0 及以上版本中，可以使用触发器来更新缓存的值，它使实现更容易。

在我们的网站例子中，例如，可以把 account_type 保存在 user 和 message 表中，而不是采用全部的非正则化。这避免了完全非正则化的插入和删除问题，因为永远不会丢失用户的信息，甚至在没有 message 的时候也不会。它不会使 user_message 表变大很多，但却让你能更高效地选择数据。

然而，现在更新用户的 account type 的代价就变高了，因为必须要更改两个表。为了证实这个问题，就必须考虑这种更新的频繁度和更新需要的时间，把它们和运行 SELECT 查询的频率进行对比。

142 另外一个好的原因是把父表的数据移到子表中以进行排序。例如，在正则化架构中按照作者的姓名对消息进行排序的代价会极高，但是可以把 author_name 缓存到 message 表中并在上面建一个索引，这样排序效率就会很高。

这对缓存衍生值也会有帮助。如果要显示每个用户发表的帖子数量（像许多论坛那样），可以运行昂贵的子查询以在每次显示的时候统计数量，也可以在 user 表上添加一个 num_message 列，并在用户发表帖子的时候更新它。

3.6.4 缓存和汇总表

有时改进性能的最佳方式是在衍生表中保持一份和父表相同的冗余数据。然而，有时须构造完全不同的汇总或缓存表，为获取数据进行特别的调优。如果可以容忍少量的陈旧数据，这种方案是最佳的，但是有时你确实没有选择（例如，你要避免复杂和昂贵的实时更新）。

“缓存表 (Cache Table)”和“汇总表 (Summary Table)”没有标准的含义。我们使用“缓存表”来指代含有能被轻易地，如果从更广泛的意义上说，则是缓慢地从架构中获取数据的表（也就是说数据是逻辑冗余的）。当说“汇总表”的时候，意味着表含有从 GROUP BY 查询取得的数据（也就是说数据不是逻辑冗余的）。一些人也会使用“上卷表 (Roll-up Table)”来指代这些表，因为数据已经被“上卷”了。

还是使用网站的例子，假设要统计过去 24 小时发布的帖子数量。在一个繁忙的站点上维护一个精确的实时计数器是不可能的。相反，应该每小时产生一个汇总表。你通常可以使用一个查询来完成这个工作，并且它比维护实时计数器效率更高。缺点就是计数器不是 100%精确的。

如果需要过去 24 小时发布的信息的精确数量，也有一个办法。从每小时的汇总表开始。然后统计给定的 24 小时之内的数量，具体方法是统计 23 个整小时，加上给定时间段开始的数量，以及给定时间段结束的数量。假设汇总表叫 msg_per_hr，并且定义如下：

```
CREATE TABLE msg_per_hr (
  hr DATETIME NOT NULL,
  cnt INT UNSIGNED NOT NULL,
  PRIMARY KEY(hr)
);
```

可以把下面 3 个查询的结果相加，得到过去 24 小时内发布消息的精确数量（注 18）：

```
mysql> SELECT SUM(cnt) FROM msg_per_hr
-> WHERE hr BETWEEN
->   CONCAT(LEFT(NOW( ), 14), '00:00') - INTERVAL 23 HOUR
->   AND CONCAT(LEFT(NOW( ), 14), '00:00') - INTERVAL 1 HOUR;
mysql> SELECT COUNT(*) FROM message
-> WHERE posted >= NOW( ) - INTERVAL 24 HOUR
->   AND posted < CONCAT(LEFT(NOW( ), 14), '00:00') - INTERVAL 23 HOUR;
mysql> SELECT COUNT(*) FROM message
-> WHERE posted >= CONCAT(LEFT(NOW( ), 14), '00:00');
```

每种方式——不精确统计或使用小范围查询以弥补时间间隔的精确统计——都比统计 message 表中全部行要高效得多，这是创建汇总表的关键原因。实时计算这些统计的代价很高，因为它们要扫描大量的数据，或者查询只有使用特别的索引才能高效运行，而这些索引并不是你想添加的，因为它们会影响更新。计算最活跃的用户或最常用的“标签 (Tags)”是这种操作的典型例子。

另一方面，缓存表对于优化搜索和获取数据的查询是有用的。这些查询通常需要特别的表和索引结构，它和普通的在线事务处理 (OLTP) 的索引结构是不同的。

例如，你也许要许多种合并的索引来加快不同类型的查询。这些冲突有时要求创建一个缓存表，它只含有主表的一部分列。一个有用的技巧是为缓存表使用不同的存储引擎。例如，如果主表使用 InnoDB，缓存表就可以使用 MyISAM，这样可以得到更小的索引并且进行全文搜索。有时你甚至想把表完全从 MySQL 中拿出来并且放到一个能更高效地进行搜索的系统中，例如 Lucene 或 Sphinx 搜索引擎。

当使用缓存和汇总表的时候，你不得不决定是否要进行实时数据维护或周期性重建。哪种方式更好取决于应用程序，但是周期性重建不仅节约资源，而且能得到没有碎片及全排序索引的更高效的表。

当重建汇总和缓存表的时候，通常数据在操作过程中还是可用的。可以通过使用“影子表 (Shadow Table)”来实现这个目标，它是构建在真实表“背后”的表。当它构建完成的时候，就可以使用原子性重命名来交换这两个表。例如，如果要重建 my_summary，就可以创建 my_summary_new，往里面填充数据，并且和真实表交换：

```
mysql> DROP TABLE IF EXISTS my_summary_new, my_summary_old;
mysql> CREATE TABLE my_summary_new LIKE my_summary;
-- populate my_summary_new as desired
mysql> RENAME TABLE my_summary TO my_summary_old, my_summary_new TO my_summary;
```

注 18：我们使用 LEFT(NOW(), 14) 把当前日期和时间圆整到最近的小时。

在把 `my_summary` 分配给新的重建表之前如果把原始的 `my_summary` 表重命名为 `my_summary_old`, 就可以在下次重建之前保持住老的版本。如果新表有问题, 则是一个方便回滚的办法。

计数表

更新计数器的时候在表中保持计数器的应用程序会遇到并发问题。这种表在网页应用程序中非常常见, 你可以使用它们来缓存用户的朋友数量, 文件下载次数等。通常建立一个单独的表, 以保持其小而快速来维护计数器, 是个好主意。使用单独的表能帮助你避免缓存失效的问题, 并且能让你使用本节中介绍的某些高级技巧。

为了尽可能保持简单, 假设有一个计数器表, 只有一行记录, 用于统计网站的点击量。

```
mysql> CREATE TABLE hit_counter (
->   cnt int unsigned not null
-> ) ENGINE=InnoDB;
```

网站的每次点击都会更新计数器:

```
mysql> UPDATE hit_counter SET cnt = cnt + 1;
```

问题是该行对于任何更新该计数器的更新事务实际上是一个全局性的“互斥量 (Mutex)”。它会序列化那些事务, 可以通过使用多行并更新随机行来得到更高的并发。这需要对表做下面的改动:

```
mysql> CREATE TABLE hit_counter (
->   slot tinyint unsigned not null primary key,
->   cnt int unsigned not null
-> ) ENGINE=InnoDB;
```

先向该表填充 100 行。现在查询可以随机地选择一行并且更新它:

```
mysql> UPDATE hit_counter SET cnt = cnt + 1 WHERE slot = RAND( ) * 100;
```

为了取得统计, 值使用聚合查询就可以了:

```
mysql> SELECT SUM(cnt) FROM hit_counter;
```

一个常见的需求是非常频繁地新开计数器 (例如, 每天一个)。如果要做到这点, 可以稍稍改动一下架构:

```
mysql> CREATE TABLE daily_hit_counter (
->   day date not null,
->   slot tinyint unsigned not null,
->   cnt int unsigned not null,
->   primary key(day, slot)
-> ) ENGINE=InnoDB;
```

这种情况下, 你不想预先产生行, 可以使用 `ON DUPLICATE KEY UPDATE`:

```
mysql> INSERT INTO daily_hit_counter(day, slot, cnt)
->   VALUES(CURRENT_DATE, RAND( ) * 100, 1)
->   ON DUPLICATE KEY UPDATE cnt = cnt + 1;
```

如果想减少行的数量, 以保持表的大小, 可以写一个周期性的任务来把所有结果合并到第一行并删除其他行:

```
mysql> UPDATE daily_hit_counter as c
->   INNER JOIN (
->     SELECT day, SUM(cnt) AS cnt, MIN(slot) AS mslot
->     FROM daily_hit_counter
->     GROUP BY day
```

```

-> ) AS x USING(day)
-> SET c.cnt = IF(c.slot = x.mslot, x.cnt, 0),
-> c.slot = IF(c.slot = x.mslot, 0, c.slot);
mysql> DELETE FROM daily_hit_counter WHERE slot <> 0 AND cnt = 0;

```

更快地读取、更慢地写入

通常需要额外的索引、冗余的字段，或者甚至缓存表和汇总表来加速读取。这为写入查询和维护工作增加了工作量，但这仍然是设计高性能查询的常见技巧：用极大地加速读取来弥补较慢写入的代价。

然而，这并不是为更快的查询付出的唯一代价，你也为读写操作增加了开发复杂性。

3.7 加速 ALTER TABLE

Speeding Up ALTER TABLE

MySQL 的 ALTER TABLE 的性能在遇到很大的表的时候会出问题。MySQL 执行大部分更改操作都是新建一个需要的结构的空表，然后把所有老的数据插入到新表中，最后删除旧表。这会耗费很多时间，尤其是在内存紧张，而表很大并含有很多索引的时候。许多人都遇到过 ALTER TABLE 操作需要几小时或几天才能完成的情况。

MySQL AB 正在改进这点。一些即将到来的改进包括不会在整个操作中锁定表的“在线 (Online)”操作。InnoDB 开发人员正在为按照排序创建索引提供支持。MyISAM 已经支持该特性，它使创建索引的速度变快了很多并且得到了更紧凑的索引布局。(InnoDB 当前是按照主键顺序一次性地构建索引，它意味着索引树不是按照优化的顺序构建的，并且被碎片化了。)

不是所有的 ALTER TABLE 操作都会导致重建表。例如，可以通过两种方式创建或去掉列的默认值（一种快，一种慢）。例如想把电影默认的租期从 3 天改为 5 天，下面是较慢的方式：

```

mysql> ALTER TABLE sakila.film
-> MODIFY COLUMN rental_duration TINYINT(3) NOT NULL DEFAULT 5;

```

使用 SHOW STATUS 分析该命令发现，它执行了 1 000 次句柄读取和 1 000 次插入。换句话说，即使列类型、大小和可空性没有变化，它也把表拷贝到了新表中。

理论上，MySQL 能跳过构建一个新表的方式。列的默认值实际保存在表的 .frm 文件中，因此可以不接触表而更改它。MySQL 没有使用这种优化，然而，任何 MODIFY COLUMN 都会导致表重建。

但是可以使用 ALTER COLUMN (注 19) 改变列的默认值：

```

mysql> ALTER TABLE sakila.film
-> ALTER COLUMN rental_duration SET DEFAULT 5;

```

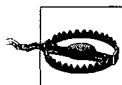
这个命令更改了 .frm 文件并且没有改动表。它非常快。

注 19：ALTER TABLE 可以使用 ALTER COLUMN、MODIFY COLUMN、CHANGE COLUMN 来修改列，每个命令做的事情都不一样。

3.7.1 只修改.frm 文件

Modifying Only the .frm File

我们已经看到修改表的.frm 文件速度很快并且 MySQL 有时会不必要地重建表。如果愿意承担一些风险，可以让 MySQL 做其他类型的修改时也不重建表。



警告：我们将要介绍的技巧是不被支持的，也没有文档记录，而且不保证一定能工作。使用它们需要自己承担风险。我们建议你在使用之前备份自己的数据。

可以执行下面的操作，而不重建表：

- 移除（但不是添加）列的 `AUTO_INCREMENT` 属性。
- 添加、移除或更改 `ENUM` 和 `SET` 常量。如果移除一个常量和一些含有该常量的行，查询将会返回空字符串。

基本的技巧是为想要的表结构创建一个.frm 文件并且把它拷贝到现有表的.frm 文件的地方，步骤如下：

1. 创建一个布局完全一样的空表，但是想改动的地方除外（例如添加的 `ENUM` 常量）。
2. 执行 `FLUSH WITH READ LOCK`。这回关闭所有正在使用的表并且防止任何表被打开。
3. 交换.frm 文件。
4. 执行 `UNLOCK TABLES` 释放读取锁。

作为例子，我们向 `sakila.Film` 表的 `rating` 列添加了一个常量，当前列像这样：

```
mysql> SHOW COLUMNS FROM sakila.film LIKE 'rating';
```

| Field | Type | Null | Key | Default | Extra |
|--------|------------------------------------|------|-----|---------|-------|
| rating | enum('G','PG','PG-13','R','NC-17') | YES | | G | |

现在为更在意电影内容的父母添加一个 PG-14 评级：

```
mysql> CREATE TABLE sakila.film_new LIKE sakila.film;
mysql> ALTER TABLE sakila.film_new
  -> MODIFY COLUMN rating ENUM('G','PG','PG-13','R','NC-17', 'PG-14')
  -> DEFAULT 'G';
mysql> FLUSH TABLES WITH READ LOCK;
```

注意到我们在常量列表的末尾添加了一个新值，如果把它放在中间，PG-13 之后，就更改变了已有数据的含义：已有 R 值就会变成 PG-14，NC-17 会变成 R，等等。

现在从系统的命令行交换.frm 文件：

```
root:/var/lib/mysql/sakila# mv film.frm film_tmp.frm
root:/var/lib/mysql/sakila# mv film_new.frm film.frm
root:/var/lib/mysql/sakila# mv film_tmp.frm film_new.frm
```

回到 MySQL 命令行，现在可以给表解锁并看看改动是否生效：

```
mysql> UNLOCK TABLES;
mysql> SHOW COLUMNS FROM sakila.film LIKE 'rating'\G
***** 1. row *****
Field: rating
Type: enum('G','PG','PG-13','R','NC-17','PG-14')
```

最后一件事情是删除用来辅助该操作的表：

```
mysql> DROP TABLE sakila.film_new;
```

3.7.2 快速建立 MyISAM 索引

快速建立 MyISAM 索引

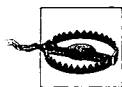
高效加载 MyISAM 表的诀窍是禁用键、加载数据、启用键：

```
mysql> ALTER TABLE test.load_data DISABLE KEYS;
-- load the data
mysql> ALTER TABLE test.load_data ENABLE KEYS;
```

这不会有问題，因为它使 MyISAM 直到所有数据被加载的时候才建立键，在这个时候，它可以按照排序构建索引。它很快并且会得到无碎片的、紧凑的索引树（注 20）。

不幸的是，它不适用于唯一索引（Unique Index），因为 DISABLE KEYS 只适用于非唯一索引。MyISAM 在内存中构建唯一索引并且在加载每一行的时候检验其唯一性。一旦索引的大小超过可用内存的时候，加载就变得极端缓慢。

就像前一节说的 ALTER TABLE 一样，如果可以承受一点点额外的工作和风险，就可以加速这个过程。比如，这对从备份中加载数据是有用的，因为你已经知道所有的数据都是有效的并且没有必要进行唯一性检查。



警告：再次提醒，这是没有文档、不受支持的技巧。使用它后果自负，并且要先备份数据。

下面是要采用的步骤：

1. 创建一个有需要的结构的表，但是没有任何索引。
2. 把数据加载到表中，以构建.MYD 文件。
3. 创建另外一个有需要的结构的表，这次包含索引。这会创建.Frm 和.MYI 文件。
4. 用读取锁刷写该表。
5. 重命名第 2 个表的.Frm 和.MYI 文件，这样 MySQL 就可以把它们用在第 1 个表上。
6. 释放掉读取锁。
7. 使用 REPAIR TABLE 创建表的索引。这会按照排序创建所有的索引，包括唯一索引。

这个过程对很大的表也很快。

3.8 对存储引擎的说明

对存储引擎的说明

本章结束部分是一些和存储引擎特定相关的说明，应当牢记这些架构设计选项。我们不想写一个巨细无遗的列表，只是描述一些和架构设计有关的关键因素。

注 20：MyISAM 在使用 LOAD DATA INFILE 和空表的时候也会按照排序创建索引。

3.8.1 MyISAM 存储引擎

The MyISAM Storage Engine

表锁

MyISAM 表有表级锁。注意不要让它成为瓶颈。

不支持自动数据恢复

如果 MySQL 服务器崩溃或掉电，就应该在使用之前进行检查和执行可能的修复。如果有大型表，这可能会花几个小时。

不支持事务

MyISAM 表不支持事务。实际上，MyISAM 甚至不保证单个命令会完成。如果在多行 UPDATE 的中途有错误发生，一些行会被更新，而另外一些行则不会。

只有索引被缓存在内存中

MyISAM 只缓存了 MySQL 进程内部的索引，并保存在键缓冲区。操作系统缓存了表的数据，因此在 MySQL5.0 中须进行昂贵的系统调用来取得它。

紧密存储

行被紧紧地保存在一起，这样磁盘上的数据就能得到小的磁盘占用和快速的全表扫描。

3.8.2 Memory 存储引擎

The Memory Storage Engine

表锁

和 MyISAM 表一样，Memory 表也有表锁。但这通常不是问题，因为 Memory 表上的查询都较快。

不支持动态行

Memory 表不支持动态(也就是可变长度)行，因此它们根本不支持 BLOB 和 TEXT 字段。即使 VARCHAR(5000) 也会变成 CHAR(5000)——如果大多数值较小，则是巨大的内存浪费。

哈希索引是默认索引类型

和其他存储引擎不同的是，如果不显式地定义，默认的索引类型是哈希类型。

没有索引统计

Memory 表不支持索引统计，因此某些复杂的查询有可能会得到不好的执行计划。

重启后丢失数据

Memory 表不会把任何数据持久到磁盘，因此，当服务器重启后即使表的定义还在，数据也会丢失。

3.8.3 InnoDB 存储引擎

The InnoDB Storage Engine

事务性

InnoDB 支持事务和四种事务隔离级别。

150

外键

在 MySQL 5.0 中, InnoDB 是唯一支持外键的存储引擎。另外的存储引擎在 CREATE TABLE 命令中可以接受外键, 但却不会强制执行。一些第三方引擎, 比如 SolidDB, 也在存储引擎层次支持它。MySQL AB 计划在未来添加服务器级别的支持。

行级锁

锁设定于行一级, 不会向上传递并且也不会阻塞选择——标准选择根本不会设定任何锁, 它有很好的并发特性。

多版本

InnoDB 使用多版本并发控制, 这样在默认情况下可能会选择读取陈旧的数据。事实上, 它的 MVCC 架构增添了很多复杂的和意料之外的行为。如果使用 InnoDB, 就应该仔细阅读 InnoDB 手册。

按主键聚集

所有的 InnoDB 表都是按主键聚集的, 可以在架构设计中运用这一点。

所有索引包含主键列

索引按照主键引用行, 因此, 如果不把主键维持得很短, 索引就增长得很大。

优化的缓存

InnoDB 把数据和内存缓存在缓冲区池里。它也会自动构建哈希索引以加快行读取。

未压缩的索引

索引没有使用前缀压缩, 因此可能会比 MyISAM 表的索引大很多。

数据装载缓慢

在 MySQL 5.0 中, InnoDB 不会特别优化数据加载。它一次构建一行的索引, 而不是按照排序进行构建。这会导致数据加载很慢。

阻塞 AUTO_INCREMENT

在 MySQL 5.1 之前的版本中, InnoDB 使用了表级锁来产生每个新的 AUTO_INCREMENT 值。

没有缓存的 COUNT(*) 值

和 MyISAM 表或 Memory 表不同, InnoDB 表不会把表的行数保存在表中, 这意味着没有 WHERE 子句的 COUNT(*) 查询不会被优化掉, 并且需要全表或索引扫描。更多细节请参阅第 188 页“优化 COUNT() 查询”。

查询性能优化

Query Performance Optimization

第 3 章讲解了如何优化数据库架构(Schema)，它是获得高性能的必要条件。但是只有架构是不够的，还需要很好地设计查询。如果查询设计得不好，那么即使是最好的架构也无法获得高性能。

查询优化、索引优化和架构优化三者相辅相成。随着设计查询的经验日益丰富，你渐渐就会明白如何用架构支撑查询；同样，你也会明白对架构的优化会影响到哪些查询。这需要时间的积累。所以当你经验更丰富的时候，不妨回到本章和第 3 章进行更深入地研读。

本章首先讨论了设计查询的基本原则，它也是查询性能不佳时最先考虑的因素；然后深入讲解了查询优化和服务器的内部机制，这部分展示了 MySQL 如何执行特定查询，从中也可以知道如何更改查询执行计划 (Query Execution Plan)；最后介绍了 MySQL 在优化查询方面的不足之处，并且探索了一些让查询获得更高执行效率的模式。

本章的目的就是让读者深入理解 MySQL 如何真正地执行查询，明白高效和低效的真正含义，在实际应用中能扬其之长，避其之短。

4.1 基本原则：优化数据访问

How Query Execution Optimizes Data Access

查询性能低下的最基本原因就是访问了太多数据。一些查询不可避免地要筛选大量的数据，但这并不常见。大部分性能欠佳的查询都可以用减少数据访问的方式进行修改。在分析性能欠佳的查询的时候，下面两个步骤比较有用：

1. 查明应用程序是否正在获取超过需要的数据。这通常意味着访问了过多的行或列。
2. 查明 MySQL 服务器是否分析了超过需要的行。

4.1.1 向服务器请求了不需要的数据？

Asking the Database for Data You Don't Want

一些查询先向服务器请求不需要的数据，然后再丢掉它们。这给服务器造成了额外的负担，增加了网络开销（注 1）(Overhead)，消耗了内存和 CPU 资源。

注 1：如果应用程序在不同的主机上，网络开销(Network Overhead)将会造成最坏的结果。但是即使 MySQL 服务器和应用程序在同一台机器上，它们之间传递数据也不是没有开销的。

下面是一些典型的错误。

提取超过需要的列

一个常见的错误就是假设 MySQL 是按要求提供结果的，而不是返回完全结果集 (Full Result Set)。这个问题在熟悉其他数据库系统的开发人员身上比较常见。他们习惯于使用 SELECT 语句选择很多行，却只提取最开始的 N 行。比如说，提取新闻网站前 100 条最新的新闻，但只在首页上显示 10 条。他们认为 MySQL 会提供这 10 条数据，然后就停止执行查询。但是 MySQL 会产生完整的计算结果，而且客户端会提取所有的数据然后把绝大部分数据都丢掉。解决这个问题的最好办法是使用 LIMIT 子句。

多表联接 (Multitable Join) 时提取所有列

如果要选择电影 Academy Dinosaur 中的所有男演员，不要使用下面的查询：

```
Mysql> SELECT * FROM sakila.actor
-> INNER JOIN sakila.film_actor USING(actor_id)
-> INNER JOIN sakila.film USING(film_id)
-> WHERE sakila.film.title = 'Academy Dinosaur';
```

这个查询返回了所有表中的所有列。正确的方式应该是这样：

```
mysql> SELECT sakila.actor.* FROM sakila.actor...;
```

提取所有的列

要对使用 SELECT* 始终持怀疑态度。真地需要所有的列吗？也许不是。获取所有列将会造成覆盖索引 (Covering Index) 这样的优化手段失效，也会增加磁盘 I/O、内存和 CPU 的开销。

基于这个原因，一些数据库管理员在所有地方都禁止使用 SELECT*，这样也可以减少由于表中列改变而造成的问题。

当然，请求超过需要的数据也不总是坏事。在我们研究过的许多案例中，开发人员告诉我们使用这种稍显浪费的方式可以简化开发，增加代码的复用性。如果明白这么做对性能的影响，那么这种做法也无可厚非。如果有其他的好想法，或者应用程序使用了某种缓存机制，那么这对获取超过实际需要的数据是很有好处的。运行大量只获取对象部分数据的单个查询时，要优先考虑获取对象的全部信息，然后缓存起来。

154

4.1.2 MySQL 检查了太多数据吗

一旦确定查询只获取了所需要的数据，那么接下来就应该检查在生成查询结果时是否检查了过多的数据。在 MySQL 中，最简单的开销指标 (Cost Metrics) 有下面 3 个：

- 执行时间
- 检查的行数
- 返回的行数

它们都不是衡量开销的完美指标，但它们大致反映了 MySQL 在内部执行查询的时候要访问多少数据，而且也大致说明了查询运行的时间。这 3 个指标都被写入了慢速查询日志 (Slow Query Log)，所以浏览该日志是检索查找了过多数据的查询的最佳方式。

执行时间

第2章已经解释了 MySQL 5.0 及其以前的版本在标准慢速查询日志上有很大的缺陷，其中包括不能进行精细记录。幸运的是，补丁可以让 MySQL 以微秒的粒度来记录查询日志。MySQL 5.1 内置了这些补丁，但以前的版本可以通过单独的补丁包得到这个功能。要明白的是不要过多地强调执行时间。它是一个不错的客观指标，但是它在不同的负载下是不一样的。另外的一些因素，比如存储引擎锁（表锁和行锁）、高并发性和硬件都对执行时间有相当大的影响。这个指标可以用于寻找对应用程序响应时间影响最大的查询，或者给服务器造成最大负载的查询，但是它不能说明某个查询在特定复杂度下执行的时间是合理的。执行时间可以是问题的表现或原因，但是这并不明显。

检查和返回的行

在分析查询的时候，考虑检查的行是有用的，因为可以从中知道找到所需要的数据的效率。

155 但是，和执行时间一样，它也不是发现问题查询的完美指标。并非所有的行访问都是一样的。较短的行有更快的访问速度，从内存中提取行数据要远快于从磁盘上提取。

在理想情况下，返回的行和检查的行应该是一样的，但是在实际中，这基本不可能。例如，使用联接来构造行时，必须要访问很多行才能产生一行输出。通常说来，检查的行和返回的行之间的比率通常较小，在 1:1 到 10:1 之间，但是偶尔它们之间也会差几个数量级。

检查的行和访问类型

在考虑查询开销的时候，想想从一个表中查找一行的情况。MySQL 使用了几种方式来返回一行，某些需要检查很多行，某些却一行都不用检查。

访问方法出现在 EXPLAIN 的 type 列，访问类型包括全表扫描（Full Table Scan）、索引扫描（Index Scan）、范围扫描（Range Scan）、唯一索引查找（Unique Index Lookup）和常量（Constant）。访问它们的速度依次递增。不需要记住所有的类型，但是应该了解扫描表、索引、范围和单个值的概念。

如果没有得到好的访问类型，那么最好的解决办法是加一个索引。第3章已经对索引做了详细的解释，现在来看看为什么索引对查询优化是如此地重要。索引让 MySQL 更有效地查找到所需要的行，访问的数据会更少。

例如，下面代码的功能是对 Sakila 数据库进行一次简单查询：

```
mysql> SELECT * FROM sakila.film_actor WHERE film_id = 1;
```

这个查询返回了 10 行结果，EXPLAIN 列出 MySQL 使用了 ref 类型访问了 idx_fk_film_id 索引。

```
mysql> EXPLAIN SELECT * FROM sakila.film_actor WHERE film_id = 1\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: film_actor
         type: ref
possible_keys: idx_fk_film_id
          key: idx_fk_film_id
        key_len: 2
         ref: const
        rows: 10
      Extra:
```

EXPLAIN 说明了它只需要访问 10 行数据, 换句话说, 查询优化器知道选定的访问类型能有效地满足查询。如果没有索引, 会出现什么样的情况? MySQL 会使用欠优化的查询, 就像下面这样:

```
mysql> ALTER TABLE sakila.film_actor DROP FOREIGN KEY fk_film_actor_film;
mysql> ALTER TABLE sakila.film_actor DROP KEY idx_fk_film_id;
mysql> EXPLAIN SELECT * FROM sakila.film_actor WHERE film_id = 1\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: film_actor
         type: ALL
possible_keys: NULL
         key: NULL
      key_len: NULL
         ref: NULL
        rows: 5073
     Extra: Using where
```

正如预料中的那样, 访问类型成了全表扫描 (All), 而且 MySQL 现在必须检查 5073 行数据。Extra 列的 “Using Where” 表明 MySQL 使用了 where 子句来过滤掉多余的行。

通常说来, MySQL 会在 3 种情况下使用 Where 子句, 从最好到最坏依次是:

- 对索引查找应用 Where 子句来消除不匹配的行, 这发生在存储引擎层。
- 使用覆盖索引 (Extra 列是 “Using Index”) 来避免访问行, 并且从索引取得数据后过滤掉不匹配的行。这发生在服务器层, 但是它不需要从表中读取行。
- 从表中检索出数据, 然后过滤掉不匹配的行 (在 Extra 列中是 “Using Where”)。这发生在服务器端并且要求在过滤之前读取这些行。

这个例子表明了好的索引多么重要。好的索引让查询有良好的访问类型并且只检查需要的行。然而添加索引并不总是意味着 MySQL 会访问并且返回同样的行。比如, 下面的代码使用了 COUNT() 函数 (注 2) 的查询。

```
mysql> SELECT actor_id, COUNT(*) FROM sakila.film_actor GROUP BY actor_id;
```

这个查询只返回 200 行, 但是它需要读取上千行数据。对于这种查询, 索引不能减少检查的行数。

不幸的是, MySQL 不会列出生成最终结果需要多少行数据, 它只会列出生成最终结果的过程中访问的总数据行数。许多数据行被 where 子句剔除了, 对最终结果没有贡献。在前一个例子中, 在移除了 sakila.film_actor 上的索引之后, 查询访问了整张表, 并且 where 子句只留下了 10 行数据。只有最后的 10 行数据用来生成最终结果。要理解服务器访问了多少行数据, 以及实际有多少行数据用于生成最终结果, 需要对查询进行分析。

如果发现访问的数据行数很大, 而生成的结果中数据行很少, 那么可以尝试更复杂的修改。

- 使用覆盖索引, 它存储了数据, 所以存储引擎不会去获取完整的行 (参阅第 3 章相关章节)。
- 更改架构, 一个例子就是使用汇总表 (Summary Table) (参阅第 3 章相关章节)。
- 重写复杂的查询, 让 MySQL 的优化器可以以优化的方式执行它 (本章稍后会讨论该话题)。

注 2: 更多信息请查阅第 188 页 “优化 COUNT()”。

4.2 重构查询的方式

Ways to Restructure Queries



当优化有问题的查询时，一个目标也许是找到一个替代的方案，但是这并不意味着要从 MySQL 得到完全一样的结果。偶尔可以用完全等价的方式得到更好的性能。但是如果不同的查询能提供更高的效率，尽管得到的结果不同，也可以考虑重写查询。也许最终应用程序的代码也会和查询一起被改写。本节说明了重构查询的技巧及使用它们的时机。

4.2.1 复杂查询和多个查询

Complex Queries Versus Many Queries

一个重要的查询设计问题就是是否可以把一个复杂查询分解成多个简单的查询。传统设计理论强调用尽可能少的查询做尽可能多的事情。这种处理方式在以前有积极意义，因为它可以节省网络通信开销，以及减少查询解析和优化的步骤。

但是，这个想法对 MySQL 不是很适用。MySQL 被设计成可以很高效地连接和断开服务器，而且能很快地响应精简的查询。现代网络比以前快了很多，延迟也小了很多。MySQL 在一般的服务器上每秒钟可以处理 50 000 个查询，如果只有一个通信对象，在千兆网络上每秒钟可以处理 2 000 个查询。因此运行多个查询并不是很糟糕的事情。

158 但是对连接的响应比起服务器内部每秒能遍历的数据行还是少很多，后者对于缓存在内存中的数据来说每秒以百万计。在其他条件相同的情况下，使用尽可能少的查询仍然是个好主意，但是有时可以通过分解查询让它得到更高的效率。不用害怕这么做，仔细地衡量开销，然后选择工作量较小的方案。本章稍后部分展示了这种技巧。

尽管如此，在应用程序中使用太多的查询是一个普遍的错误。例如，一些应用程序使用了 10 条单行查询从一个表中取数据，但实际可以使用一条语句来代替它。我们曾经看到程序为了取单个列，对同一行操作了很多次！

4.2.2 缩短查询

Chopping Up a Query

一种处理查询的办法是分治法，让查询在本质上不变，但是每次只执行一小部分，以减少受影响行数。

清理陈旧数据是一个很好的例子。周期性的清理工作需要移除大量的数据，如果用一个很大的查询来做这个工作，就会长时间地锁住很多行数据，塞满事务日志，耗尽资源，打断一些本不该被打断的查询。采用细化 DELETE 语句并使用中等大小的查询会极大地改进性能，并且在复制的时候减少延迟。比如，下面是一个巨大的查询：

```
mysql> DELETE FROM messages WHERE created < DATE_SUB(NOW( ), INTERVAL 3 MONTH);
```

应该用类似于下面的伪代码的查询代替它。

```
rows_affected = 0
do {
    rows_affected = do_query(
        "DELETE FROM messages WHERE created < DATE_SUB(NOW( ), INTERVAL 3 MONTH)
        LIMIT 10000")
} while rows_affected > 0
```

对于一个高效的查询来说，一次删除 10 000 行数据的任务已经足够大了。足够短的任务对服务器的影响最小（注 3）（事务存储引擎会从较小的事务中得益）。在 DELETE 语句中加入休眠语句也是一个好主意，它可以分摊负载，并且减少锁住资源的时间。

4.2.3 分解联接

mysql> SELECT * FROM tag

许多高性能的网站都用了“分解联接”技术，可以把一个多表联接分解成多个单个查询，然后在应用程序端实现联接操作，比如下面的多表联接语句：

```
mysql> SELECT * FROM tag
-> JOIN tag_post ON tag_post.tag_id=tag.id
-> JOIN post ON tag_post.post_id=post.id
-> WHERE tag.tag='mysql';
```

可以用下面的语句代替

```
mysql> SELECT * FROM tag WHERE tag='mysql';
mysql> SELECT * FROM tag_post WHERE tag_id=1234;
mysql> SELECT * FROM post WHERE post.id in (123,456,567,9098,8904);
```

第一眼看上去这比较浪费，因为这只是增加了查询的数量而已。但是这种重构方式有下面这些重大的性能优势：

- 缓存的效率更高。许多应用程序都直接缓存了表。在这个例子中，如果有 mysql 这个标签的对象已经被缓存了，那么第一个查询就可以跳过。如果在缓存中找到了 id 为 123、567 或 9098 的信息，那么就可以从 IN() 中把它们移除。查询缓存也可以从中得益。如果只有一个表经常改变，那么分解联接就可以减少缓存失效的次数。
- 对 MyISAM 表来说，每个表一个查询可以更有效地利用表锁，因为查询会锁住单个表较短时间，而不是把所有表长时间锁住。
- 在应用程序端进行联接可以更方便地扩展数据库，把不同的表放在不同的服务器上。
- 查询本身会更高效。在这个例子中，使用 IN() 而不是联接让 MySQL 可以对行 ID 进行排序，并且更高效地取得数据。稍后会介绍这部分的详情。
- 可以减少多余的行访问。在应用程序端进行联接意味着对每行数据只会访问一次，而联接从本质上来说是非正则化（Denormalization）的，它会反复地访问同一行数据。基于同样的原因，这种重构方式可以减少网络流量和内存消耗。
- 在某种意义上，你可以认为这种方式是手工执行了哈希联接，而不是 MySQL 内部执行联接操作时采用的嵌套循环算法。哈希联接效率更高（本章稍后会讨论 MySQL 的联接策略）。

注 3：Maatkit 的 mk-archiver 工具能简化这些工作。

小结：什么时候在应用程序端进行联接效率更高

在下面的场景中，在应用程序端进行联接效率更高：

- 可以缓存早期查询的大量数据。
- 使用了多个 MyISAM 表。
- 数据分布在不同的服务器上。
- 对于大表使用 `IN()` 替换联接。
- 一个联接引用了同一个表很多次。

4.3 查询执行基础知识

Query Execution Basics

想得到高性能，最佳的方式就是学习 MySQL 如何优化和执行查询。一旦理解了背后的机制，那么很多优化工作就简化为纯粹的推理，并且也可以理解查询优化过程中的逻辑性。



提示：本节假设你已经阅读过第 2 章，它提供了 MySQL 查询执行引擎的基础知识。

图 4-1 显示了 MySQL 执行查询的一般性过程。

下面对该过程进行一个简述：

1. 客户端将查询发送到服务器。
2. 服务器检查查询缓存。如果找到了，就从缓存中返回结果，否则进行下一步。
3. 服务器解析，预处理和优化查询，生成执行计划。
4. 执行引擎调用存储引擎 API 执行查询。
5. 服务器将结果发送回客户端。

上面的每一步都有一些额外的复杂性，接下来将讲解这些过程，同时阐述每一步对应的状态。要知道，查询优化的过程尤其复杂和重要。

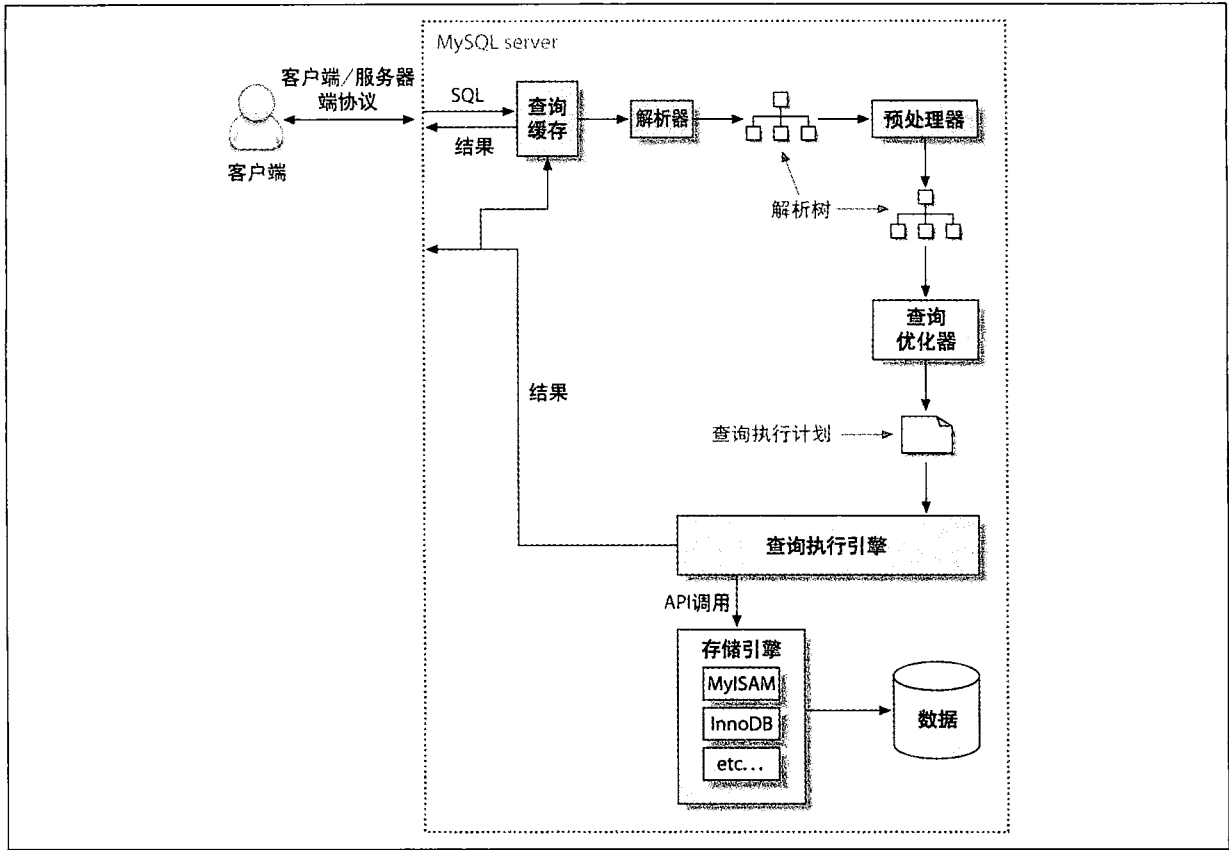


图 4-1: 查询的执行路径

4.3.1 MySQL 客户端/服务器协议

尽管不用理解 MySQL 客户端/服务器协议的内部细节，但是还是需要了解其大致轮廓。这个协议是半双工的，这意味着 MySQL 服务器在某个给定的时间，可以发送或接收数据，但是不能同时发送和接收。这也意味着没有办法截断消息。

这种协议让 MySQL 的沟通简单而又快捷，但是它也有一些限制。其中一个就是无法进行流程控制，一旦一方发送消息，另一方在发送回复之前就必须提取完整的消息。这像来回抛球的游戏：在任意时刻，只有某一方有球，而且除非有球在手上，否则就不能把球抛回去（发送消息）。

客户端用一个数据包将查询发送到服务器。这就是为什么 `max_packet_size` 这个配置参数对于大查询（注 4）很重要的原因。一旦客户端发送了查询，那就意味着“球”已经不在自己手上了，唯一能做的就是等待结果。

但是，服务器发送的响应由许多数据包组成。服务器发送响应的时候，客户端就必须接收完整的结果集。它不能只提取几行数据后就要求服务器停止发送剩下的数据。如果客户端只需要其中的几行数据，要么等待所有数据都传送完毕后丢掉不用的数据，要么就笨拙地断开连接。这两种办法都不好，这就是为什么 `LIMIT` 子句很重

注 4：如果查询过大，那么服务器会拒绝接收数据并且抛出一个错误。

要的原因。

还有另外一种理解方式，当客户端从服务器提取数据的时候，它认为所有数据都是从服务器“拉”过来的，但实际情况是服务器在产生这些数据的同时就把它们“推”到客户端。客户端只需要接收推出来的数据，根本就没办法告诉服务器停止发送数据。

绝大多数连接 MySQL 的类库能让你提取完整的结果然后缓存到内存中，或者只提取需要的数据。默认的行为通常是提取所有数据然后缓存。这很重要，因为 MySQL 只有在所有数据都被提取之后才会释放掉所有的锁和资源。查询的状态会是“发送数据”（参阅第 163 页的“查询状态”）。如果客户端类库一次性提取了所有的数据，那么就可以减少服务器所做的工作，让服务器可以尽可能快地完成所有的清理工作。

大部分客户端类库可以让使用者像直接从服务器上提取数据一样处理结果，但是它实际上只是在类库的内存中处理数据。这种机制在大多数时候都工作良好，但是对于很庞大的结果集也许会需要很长的时间和很多内存。如果不缓存数据，那么就可以使用较少的内存，并且尽快开始工作。这么做的缺点就是在应用程序和类库交互的时候，服务器端的锁和资源都是被锁定的（注 5）。

下面是一个 PHP 的例子，首先，是在 PHP 中使用 MySQL 的惯用方式：

```
<?php
$link = mysql_connect('localhost', 'user', 'p4ssword');
$result = mysql_query('SELECT * FROM HUGE_TABLE', $link);
while ( $row = mysql_fetch_array($result) ) {
    // Do something with result
}
?>
```

163 在 while 循环中，看上去好像只是在需要的时候才从数据库提取数据，但实际上代码利用 `mysql_query()` 从缓存中提取数据。While 循环只是简单地遍历缓存。下面的代码不会缓存数据，它使用的是 `mysql_unbuffered_query()`：

```
<?php
$link = mysql_connect('localhost', 'user', 'p4ssword');
$result = mysql_unbuffered_query('SELECT * FROM HUGE_TABLE', $link);
while ( $row = mysql_fetch_array($result) ) {
    // Do something with result
}
?>
```

编程语言有不同的方式来处理缓存。比如，Perl 的 DBD::mysql 驱动会要求定义 C 客户端库的 `mysql_use_result` 属性，默认值是 `mysql_buffer_result`，下面是一个示例。

```
#!/usr/bin/perl
use DBI;
my $dbh = DBI->connect('DBI:mysql;host=localhost', 'user', 'p4ssword');
my $sth = $dbh->prepare('SELECT * FROM HUGE_TABLE', { mysql_use_result => 1 });
$sth->execute();
while ( my $row = $sth->fetchrow_array() ) {
    # Do something with result
}
```

注意，调用 `prepare()` 函数让结果不缓存。也可以在连接的时候定义这个参数，这样每条语句都是不缓存的。

```
my $dbh = DBI->connect('DBI:mysql:mysql_use_result=1', 'user', 'p4ssword');
```

注 5：也可以用 `SQL_BUFFER_RESULT`，稍后会讨论这一话题。

查询状态

每个 MySQL 连接，或者叫线程（Thread），在任意一个给定的时间都有一个状态来标识正在进行的事情。有几种方法可以查看状态，但是最容易地是使用 `SHOW FULL PROCESSLIST` 命令（状态会出现在 `Command` 列）。当一个查询处于其生命周期中的时候，它的状态会改变多次。一共有 12 种状态。MySQL 手册定义了所有状态。这儿只列出了其中几种状态并解释了其含义。

休眠

线程正在等待客户端的新查询。

查询

线程正在执行查询或往客户端发送数据。

锁定

线程正在等待服务器授予一个表锁。由存储引擎执行的锁，比如 InnoDB 的行锁，不会让线程进入 `Locked` 状态。

分析和统计

线程正在检查存储引擎的统计信息并且优化查询。

拷贝到磁盘上的临时表

线程正在处理查询并且把结果拷贝到暂存表，其结果也许是 `GROUP BY`、对文件排序、使用 `UNION`。如果状态后面有“on disk”，那么说明 MySQL 正在把内存中的表拷贝到磁盘上的表里面。

排序结果

线程正在对结果进行排序。

发送数据

这有几种含义。线程也许是在查询的各个状态之间传递数据，也可能是在产生结果，还有可能是把结果返回给客户端。

知道基本状态是很有帮助的，这样可以知道“球在谁手上”。在非常繁忙的服务器上，有可能看到不常见的状态，或者常见的简报状态（比如 `statistics`）占据了很长的时间。这通常意味着有问题发生。

4.3.2 查询缓存

The Query Cache

在解析一个查询之前，如果开启了缓存，MySQL 会检查查询缓存，进行大小写敏感的哈希查找。即使查询和缓存中的查询只有一个字节的差异，也表示不匹配。查询就会进入到下一个状态。

如果 MySQL 在缓存中发现了一个匹配，那么在返回缓存之前，必须检查查询的权限。因为 MySQL 在缓存中存储了表的信息，所以这时可能根本不用解析查询。如果权限没有问题，那么 MySQL 就从缓存中提取数据，然后返回给客户端。这样就绕过了查询执行的所有步骤，不需要对查询进行解析、优化和执行。

第 5 章详细介绍了查询缓存。

4.3.3 查询优化过程

The Query (and optimization) process

查询生命周期的下一步是把查询转变成执行计划。它有几个子过程：解析、预处理和优化。错误（比如语法错误）在这个过程中的任何一步都可能出现。在这儿不是详细介绍 MySQL 的内部机制，所以可以用比较自由的方式来介绍这个过程。尽管下面的步骤在实际中出于效率的考虑，是合并在一起的，但是我们还是分开介绍它们，目的是帮助你理解 MySQL 如何执行查询，让你可以写出更好的查询。

解析器和预处理器

首先，MySQL 解析器将查询分解成一个个标识（Token），然后构造一棵“解析树（Parse Tree）”。解析器使用 MySQL 的语法规则解析和验证查询。比如，它保证查询中的标识都是有效的，并且在适当的位置上。它也会检查其中的错误，比如字符串上面的引号没有闭合。

然后，预处理器（Preprocessor）检查解析器生成的结果树，解决解析器无法解析的语义。比如，它会检查表和列是否存在，它也会检查名字和别名，确保引用没有歧义。

最后，预处理器检查权限。这通常是很快的过程，除非机器上有大量的权限（第 12 章有更多关于权限和安全的内容）

查询优化器

解析树现在已经通过了验证，并且准备让优化器把它变成执行计划。一个查询通常可以有很多种执行方式，并且返回同样的结果，优化器的任务就是找到最好的方式。

MySQL 使用的是基于开销（Cost）的优化器，这意味着它会预测不同执行计划的开销，并且选择开销最小的一个。开销的单位是一次对大小为 4KB 的页面的随机读取。下面是一个使用 `Last_query_cost` 变量来了解查询开销的例子：

```
mysql> SELECT SQL_NO_CACHE COUNT(*) FROM sakila.film_actor;
+-----+
| count(*) |
+-----+
|    5462  |
+-----+
mysql> SHOW STATUS LIKE 'last_query_cost';
+-----+-----+
| Variable_name | Value          |
+-----+-----+
| Last_query_cost | 1040.599000    |
+-----+-----+
```

结果表示优化器认为这个查询会造成 1040 次随机读取。它来自对统计数据的估计，这些统计数据包括每个表或索引的页面数量、索引的基数性（Cardinality）（译注 1）、键和行的长度，以及键的分布。优化器不会考虑任何缓存因素，它认为每次读取都会有相同的 IO 开销。

译注 1：基数性（Cardinality）用于衡量表中某列数据之间的唯一性。基数性越低，说明该列数据的重复性越高，反之则说明数据的重复性越低。

由于种种原因，优化器并不总是能选择最好的方案：

- 统计数据可能是错误的。服务器依赖于存储引擎提供的统计，它可能非常准确，也可能非常不准确，比如 InnoDB 存储引擎因为 MVCC 架构的缘故不会维护表行数的精确统计。
- 开销指标和运行查询的实际开销并不精确地相等。所以即使统计数据是准确的，查询的开销也可能和 MySQL 的估计不一致。一个读取更多页面的查询在某些情况下开销可能更小，比如读取磁盘上的顺序数据时 I/O 效率更高，或者数据已经被缓存在内存中。
- MySQL 的优化并总是和我们想的一样。我们有时候需要更快的执行时间，但是 MySQL 并不真正地理解“快”的含义，它只考虑开销。就像看到的那样，它对开销的预计也不是完全精确的。
- MySQL 不会考虑正在并发运行的其他查询，而并发查询会影响查询运行的速度。
- MySQL 并不总是根据开销来进行优化。有时候它仅仅遵从一些原则，比如：“如果这儿有一个全文 MATCH() 子句，并且存在 FULLTEXT 索引，那么就使用它”。即使使用不同的索引，或者使用带 WHERE 子句的非 FULLTEXT 查询更快的情况下也是如此。
- 优化器不会考虑不受它控制的操作的开销，比如执行存储函数和用户定义的函数。
- 稍后会看到，优化器不会总是估算每一个可能的执行计划，所以它可能会错过优化方案。

MySQL 的优化器是相当复杂的，它使用了很多优化技巧把查询转换为执行计划。有两种基本的优化方案：**静态优化和动态优化**。静态优化可以简单地通过探测解析树 (Parse Tree) 来完成。比如，优化器可以通过运用代数化法则 (Algebraic Rules) 把 WHERE 子句转换成相等的形式。静态优化和值无关，比如 WHERE 子句中的常量。它可以被应用一次，然后始终都有效，即使用不同的参数重新执行查询也不会改变。可以把这个优化看成是“编译时优化”。

相反，**动态优化**根据上下文而定，和很多因素有关，比如 WHERE 子句中的值和索引中的行数。每次查询执行的时候都会重新执行优化，可以把它看成“运行时优化”。

执行 MySQL 语句和存储过程的区别是很重要的。MySQL 只执行一次静态优化，然后每次运行查询的时候都会进行动态优化。MySQL 有时甚至在执行的时候还会进行优化（注 6）。

下面是 MySQL 能够处理的一些优化类型：

对联接中的表重新排序

表并不总是按照查询中定义的顺序进行联接，决定最佳的联接顺序是一种重要的优化。第 173 页深入阐述了这一主题。

将外联接转换成内联接

外联接并不总是按照外联接的方式执行。一些因素，比如 WHERE 子句和表的架构，都能够将外联接转化等价的内联接。MySQL 能区分这些情况并且重写联接，使之适合重新排序。

代数等价法则

MySQL 使用代数化转换来简化并且规范化表达式。它可以隐藏或减少常量，移除不可能的限制和常量条

注 6：比如，使用联接时范围检查查询计划会为每一行计算索引。可以在 EXPLAIN 的 Extra 列中看到“为每一条记录执行范围检查”。这个查询计划也会增加 Select_full_range_join 这个服务器变量的值。

件。比如 $5=5 \text{ AND } a>5$ 就会被精简成 $a>5$ 。 $(a<b \text{ AND } b=c) \text{ AND } a=5$ 被转化成 $b>5 \text{ AND } b=c \text{ AND } a=5$ 。这些规则在重写条件查询的时候非常有用，稍后会讨论这一话题。

优化 COUNT(), MIN() 和 MAX()

索引和列的可空性常常能帮助 MySQL 优化掉这些表达式。比如，为了查找一个位于 B 树最左边的列的最小值，那么直接找索引的第一行就可以了。这种优化即使在查询优化的阶段也可以发生。同样地，为了查找 B 树索引中的最大值，那么找最后一行就可以了。如果服务器使用了这种优化，那么就可以在 EXPLAIN 中看到“选择被优化掉的表”。它字面意义是表从查询计划中被移走了，代替它的是一个常数。

同样地，没有 where 子句的 COUNT(*) 通常也会被一些存储引擎优化掉（比如 MyISAM 总是保留着表行数的精确值）。本章稍后的第 188 页的“优化 COUNT() 查询”更详细地讨论了该话题。

计算和减少常量表达式

如果 MySQL 探测到一个表达式可以被简化为一个常量，那么它就会在优化期间做这种转换。比如说，一个用户定义的变量如果没有变化，那么它就可以被转化为一个常量。算术表达式是另外一个例子。

也许会让人很惊讶的是，一些查询语句在优化过程中也会被简化为常量。一个例子就是应用在索引上的 MIN() 函数。它可以被扩展成在主键或唯一索引上的常量查找。如果 WHERE 子句对索引使用了常量条件，那么优化器在查询开始的时候就可以查找它的值，然后在查询的剩余部分把它当成常量。下面是一个例子：

168

```
mysql> EXPLAIN SELECT film.film_id, film_actor.actor_id
-> FROM sakila.film
-> INNER JOIN sakila.film_actor USING(film_id)
-> WHERE film.film_id = 1;

+----+-----+-----+-----+-----+-----+-----+
| id | select_type | table      | type | key           | ref      | rows |
+----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE      | film       | const | PRIMARY      | const    | 1    |
| 1  | SIMPLE      | film_actor | ref   | idx_fk_film_id | const    | 10   |
+----+-----+-----+-----+-----+-----+-----+
```

MySQL 分两步执行该查询，它们分别对应结果中的两行。第 1 步是在 film 表中找到需要的行。因为 film_id 列有主键，所以优化器在优化过程中就可以查询索引，然后知道要找的数据只有 1 行。因为优化器用一个已知的数据去进行查找，所以表的 ref 类型就是 const。

第 2 步中，因为优化器知道所有从第一步来的数据，所以它可以把第一步查找到的 film_id 列当成已知量。要注意，film_actor 表的 ref 类型也是 const，和 film 表一样。

在 WHERE、USING、ON 这些强制值相等的子句中，常量具有传递性，这也是应用常量条件的一种情况。在本例中，优化器知道 USING 子句强制 film_id 在查询中的所有地方都是相等的，它必须等于 WHERE 子句中的常量值。

覆盖索引

当索引包含查询需要的所有列时，MySQL 有时可以使用索引来避免读取行数据。在第 3 章中详细讨论了覆盖索引。

子查询优化

MySQL 可以将某些类型的子查询转换成相等的效率更高的形式，把它们简化为索引查找，而不是独立的

多个查询。

早期终结

一旦满足查询或某个步骤的条件，MySQL 就会立即停止处理该查询，或者该步骤。LIMIT 子句是一个明显的例子。另外还有一些其他的终结方式。比如，MySQL 检查到一个不可能的条件，它就会停止整个查询。下面是一个例子：

```
mysql> EXPLAIN SELECT film.film_id FROM sakila.film WHERE film_id = -1;
+----+ +...+-----+
| id  | |...| Extra |
+----+ +...+-----+
| 1   | |...| Impossible WHERE noticed after reading const tables |
+----+ +...+-----+
```

这个查询在优化阶段就停止了。在某些情况下，MySQL 也能很快地停止查询。比如当执行引擎知道需要取得唯一的值，或者值不存在的时候，服务器就可以使用这种优化手段。比如，下面的查询试图查找一部没有男演员的电影（注 7）。

```
mysql> SELECT film.film_id
-> FROM sakila.film
-> LEFT OUTER JOIN sakila.film_actor USING(film_id)
-> WHERE film_actor.film_id IS NULL;
```

这个查询的策略是剔除有男演员的电影。每部电影可能都有很多男演员，但是只要发现有一名男演员，它就立即停止处理这部电影，然后马上转向下一部，因为执行引擎知道 WHERE 子句不允许这部电影出现在结果中。类似的“不同值/不存在 (DISTINCT/NOT-EXISTS)”优化方式可以应用到某些使用了 DISTINCT、NOT EXISTS() 和 LEFT JOIN 的查询中。

相等传递

MySQL 能辨认一个查询中有两个列相等的情况。比如，在 JOIN 和 WHERE 子句之间使用相等的列，就像下面这样：

```
mysql> SELECT film.film_id
-> FROM sakila.film
-> INNER JOIN sakila.film_actor USING(film_id)
-> WHERE film.film_id > 500;
```

MySQL 知道 WHERE 子句不仅被应用到 film 表，也被应用到了 film_actor 表，因为 USING 子句强制两个表中的 film_id 列相等。

如果其他的数据库服务器不支持这种方式，那么就需要自己帮助优化器优化查询，那么 WHERE 子句就可能像下面这样：

```
WHERE film.film_id > 500 AND film_actor.film_id > 500
```

MySQL 里面不需要这么做，这只会让查询更难维护。

注 7：大家都同意一部电影没有男演员是很奇怪的。但是 Sakila 数据库里面确实有一部电影没有男演员，它就是“SLACKER LIAISONS”，可以简要地把它概括为“一条鲨鱼和一名必须要到古代中国和一条鳄鱼见面的学生之间发生的快节奏故事。”

比较 IN() 里面的数据

许多数据库服务器都只是把 IN() 看作多个 OR 的同义词，因为它们在逻辑上是相等的。MySQL 不是这样，它会对 IN() 里面的数据进行排序，然后用二分法查找某个值是否在列表中，这个算法的效率是 $O(\log n)$ ，而等价的 OR 子句的查找效率是 $O(n)$ 。在列表很大的时候，OR 子句就会慢得多。

上面的列表并不完整。MySQL 执行了很多优化，即使用整整一章的篇幅来描述也不够。但是这些例子可以让你明白优化器的复杂性和智能性。如果这些讨论只能有一个结论，那就是“不要试着比优化器更聪明”。每个人都可能想出种种办法来打败优化器，但这只能让自己的查询变得更复杂、更难以维护，却没有丝毫的好处。通常说来，应该让优化器按照自己的方式来优化查询。

当然，尽管优化器很智能，但它有时候也不能给出最佳结果。有时候你比优化器更了解数据，比如因为应用程序的逻辑，某些判断必定为真。同样，优化器有时候也没有需要的功能，比如哈希索引。再者，就像前面所说的那样，它选择的方案的开销也许比替代方案更高。

如果知道优化器没有给出好的结果，而且知道为什么，那么就可以帮助优化器做更多优化。可以把一些选项加到查询里面作为给优化器的提示，也可以重写查询，重新设计数据库架构或加上索引。

表和索引统计

回想一下图 1-1 中 MySQL 服务器架构的各种层次。服务器层有查询优化器，却没有保存数据和索引的统计数据。存储引擎负责统计这些数据，每个存储引擎都应该包含各种统计数据（也可能是以不同的方式把它们保存在一个存储引擎中）。诸如 Archive 这样的引擎根本就不保存任何数据！

因为服务器没有保存统计数据，所以 MySQL 优化器就不得不向存储引擎询问查询所使用的表的统计数据。存储引擎会向优化器提供每个表或索引的页面数量、表和索引的基数性、键和行的长度及键的分布信息。优化器可以使用这些信息来决定最佳的执行计划。

后续章节将会讲解统计数据如何影响优化器选择优化方案。

MySQL 的联接执行策略

MySQL 在执行过程中非常广泛地使用了“联接”这一术语。总地说来，它将每个查询都看成一个联接，这并不仅仅是指在两个表中查找匹配的查询，而是指每一个查询语句，包括子查询，甚或针对单个表的 SELECT 语句。这样说来，理解 MySQL 如何执行联接自然也显得非常重要。

考虑一下联合 (UNION) 查询。MySQL 将 UNION 看成一系列的单个查询，它们将结果写入临时表中，最后再读出来组成最终结果。在 MySQL 中，每个单个查询都是一个联接，所以从临时表读取数据实际也是联接。

MySQL 的联接执行策略很简单，它把每个联接都看成一个嵌套循环，这意味着 MySQL 用一个循环从表中读取数据，然后再利用一个嵌套循环从下一个表中发现匹配数据。它不停地持续这个过程，当发现一行匹配的数据时，再根据 SELECT 子句中的列输出。接着它会试着查找更多匹配的行，如果没有找到，就回溯到前一个表，用新的行开始新一轮迭代。如果前一个表中没有行了，它就会不停地回溯，直到有新的可用行为止。这时候，它按照上面的嵌套循环过程在下一个表中查找匹配数据，依次执行迭代（注 8）。

注 8：稍后会看到 MySQL 执行查询并不是如此简单，很多优化措施让它变得很复杂。

查找行、探测下一个表、回溯的过程可以写成一个嵌套循环，因此可以把它叫做“嵌套循环联接”，下面是一个简单的例子：

```
mysql> SELECT tbl1.col1, tbl2.col2
-> FROM tbl1 INNER JOIN tbl2 USING(col3)
-> WHERE tbl1.col1 IN(5,6);
```

假设 MySQL 不会改变查询中表的顺序，那么 MySQL 执行查询的伪代码大致如下：

```
outer_iter = iterator over tbl1 where col1 IN(5,6)
outer_row = outer_iter.next
while outer_row
    inner_iter = iterator over tbl2 where col3 = outer_row.col3
    inner_row = inner_iter.next
    while inner_row
        output [ outer_row.col1, inner_row.col2 ]
        inner_row = inner_iter.next
    end
    outer_row = outer_iter.next
End
```

这个执行计划适用于单表查询和多表查询，这也是为什么单表查询可以被看成联接的原因。单表联接是多表复杂联接的基础，它也能支持外联接（OUTER JOIN）。如果把查询改成下面这样：

```
mysql> SELECT tbl1.col1, tbl2.col2
-> FROM tbl1 LEFT OUTER JOIN tbl2 USING(col3)
-> WHERE tbl1.col1 IN(5,6);
```

相应的伪代码如下：

```
outer_iter = iterator over tbl1 where col1 IN(5,6)
outer_row = outer_iter.next
while outer_row
    inner_iter = iterator over tbl2 where col3 = outer_row.col3
    inner_row = inner_iter.next
    if inner_row
        while inner_row
            output [ outer_row.col1, inner_row.col2 ]
            inner_row = inner_iter.next
        end
    else
        output [ outer_row.col1, NULL ]
    end
    outer_row = outer_iter.Next
End
```

172

另一种形象地显示执行计划的方式是“泳道图（Swim-Lane Diagram）”。如图 4-2 所示。它显示了上面采用内联接（INNER JOIN）的例子的执行计划。请按照从上到下、从左到右的顺序阅读“泳道图”。

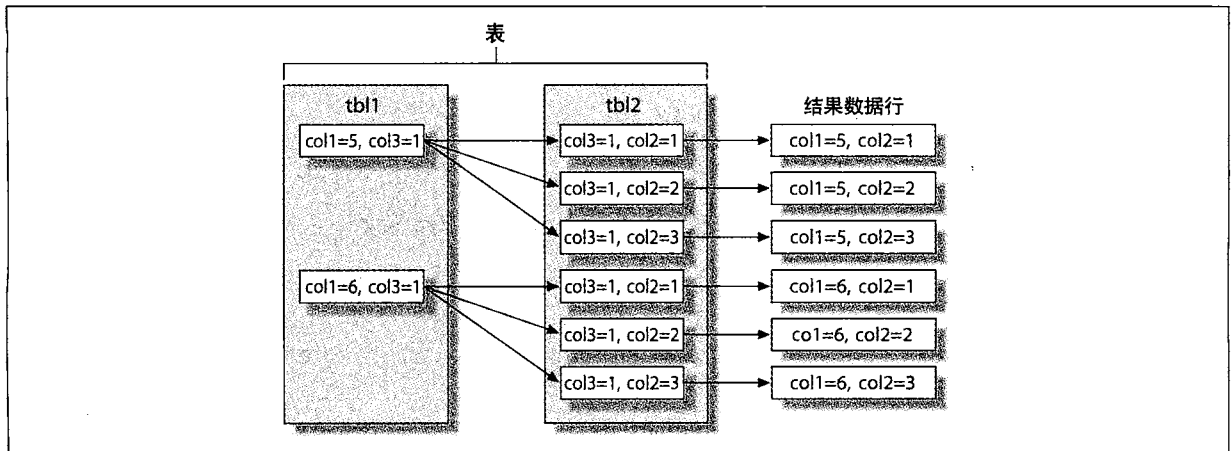


图 4-2: “泳道图”显示了如何使用联接取得数据

从本质上说，MySQL 以同样的方式执行每一种查询。例如，在处理 FROM 子句中的子查询时，它会先执行子查询，并且把结果放到临时表（注 9）里面，然后把临时表当成普通表进行下一步处理（因而叫它“衍生表（Derived Table）”）。MySQL 也使用临时表来处理联合（UNION），它还会把所有的右外联接（RIGHT OUTER JOIN）改写成等价的左外联接（LEFT OUTER JOIN），简言之，MySQL 把所有的查询都强行转换为联接来执行。

但是这种方式并不是对所有的合法查询都有效。比如，全外联接（FULL OUTER JOIN）不能用嵌套循环，以及没发现匹配的数据时进行回溯的方式来执行，原因是检索的第一个表内可能根本就没有匹配的数据。这也解释了 MySQL 为什么不支持全外联接。某些查询可以用嵌套循环的方式来执行，但是效率极差。后文会对其中一些情况进行解释。

执行计划

MySQL 不会产生字节码（Byte-Code）来执行查询，这和其他许多数据库不一样。实际上，MySQL 执行计划是树形结构，目的是指导执行引擎产生结果。最终的计划中包含了足够的信息来重建查询。如果对某个查询使用 EXPLAIN EXTENDED 命令，并加上 SHOW WARNINGS 参数，就可以看到重建后的查询（注 10）。

从概念上说，任何多表查询都用树（Tree）来表示。例如，一个四表查询的执行计划可能如图 4-3 所示。

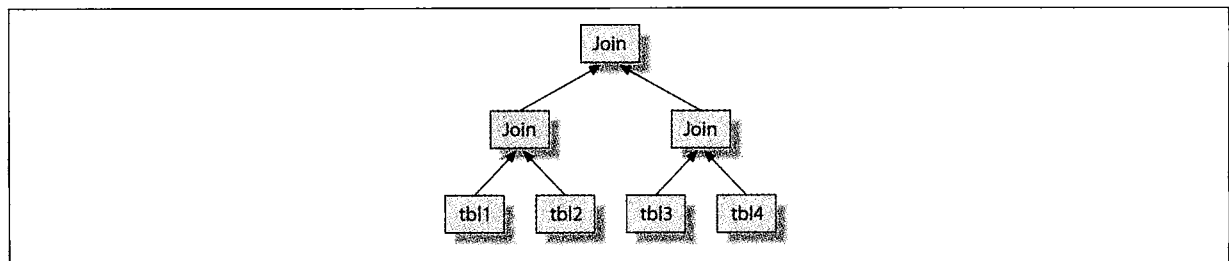


图 4-3: 一种多表联接的方式

这是所谓的“平衡树（Balanced Tree）”。但 MySQL 不会按这样的方式来执行。在上一节中已经说过，MySQL

注 9: 临时表没有索引，在决定 FROM 子句中到底是用复杂的联接还是用子查询时要牢记这一点。联合也是一样的。

注 10: 服务器从执行计划产生结果。因此它和原始的查询计划有相同的语义，但是并不一定在文字上完全一样。

总是从一个表开始，然后在下一个表中寻找匹配的行。因此，MySQL 的执行计划总是采用“左深度树 (Left Deep Tree)”，如图 4-4 所示。

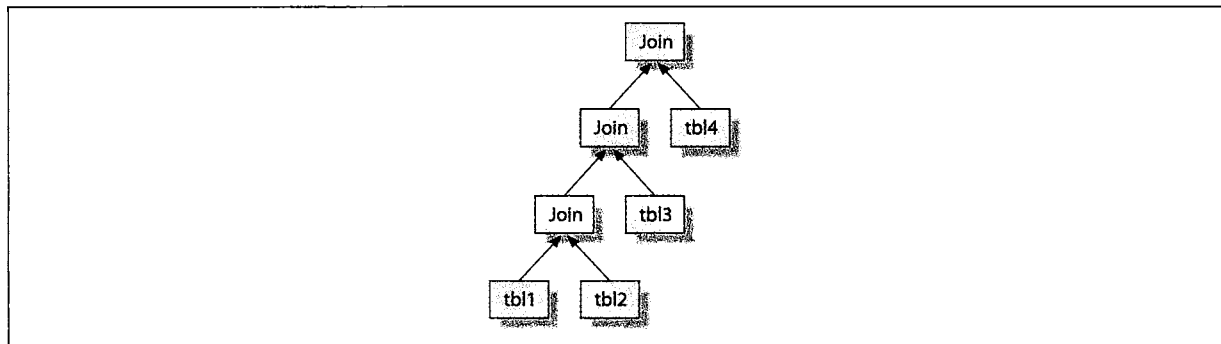


图 4-4: MySQL 进行多表联接的方式

联接优化器

MySQL 优化器中最重要的部分是**联接优化器**，它决定了多表查询的最佳执行顺序。通常可以用不同的顺序联接几个表，然后得到同样的结果。联接优化器评估不同执行计划的开销，并且选择开销最低的计划。

下面的查询可以用不同的顺序联接表，但结果都是一样的：

```
mysql> SELECT film.film_id, film.title, film.release_year, actor.actor_id,
->      actor.first_name, actor.last_name
->      FROM sakila.film
->      INNER JOIN sakila.film_actor USING(film_id)
->      INNER JOIN sakila.actor USING(actor_id);
```

这个查询可以有不同的执行。比如，可以从 film 表开始，使用 film_actor 表中 film_id 列的索引来查找 actor_id 的值，然后再检索 actor 表的主键以得到需要的数据。这听上去是很有效率的，不是吗？现在使用 EXPLAIN 看看 MySQL 是如何执行这个查询的：

```
***** 1. row *****
  id:      1
select_type:  SIMPLE
  table:    actor
  type:     ALL
possible_keys: PRIMARY
  key:      NULL
  key_len:  NULL
  ref:      NULL
  rows:     200
  Extra:
***** 2. row *****
  id:      1
select_type:  SIMPLE
  table:    film_actor
  type:     ref
possible_keys: PRIMARY,idx_fk_film_id
  key:      PRIMARY
  key_len:  2
  ref:      sakila.actor.actor_id
  rows:     1
  Extra:    Using index
```

174

```

***** 3. row *****
      id:      1
  select_type: SIMPLE
        table: film
         type:  eq_ref
possible_keys: PRIMARY
         key:   PRIMARY
        key_len: 2
         ref:   sakila.film_actor.film_id
         rows:  1
       Extra:

```

这和预料的完全不一样。MySQL 从 actor 表开始（它位于 EXPLAIN 输出的第一行），执行的顺序和预料的相反。这样做真地效率更高么？让我们来检查一下，关键字 STRAIGHT_JOIN 强制执行引擎按照查询中表出现的顺序来进行联接操作。下面是 EXPLAIN 的输出：

```

mysql> EXPLAIN SELECT STRAIGHT_JOIN film.film_id...\G
***** 1. row *****
      id:      1
  select_type: SIMPLE
        table: film
         type:  ALL
possible_keys: PRIMARY
         key:   NULL
        key_len: NULL
         ref:   NULL
         rows:  951
       Extra:
***** 2. row *****
      id:      1
  select_type: SIMPLE
        table: film_actor
         type:  ref
possible_keys: PRIMARY,idx_fk_film_id
         key:   idx_fk_film_id
        key_len: 2
         ref:   sakila.film.film_id
         rows:  1
       Extra: Using index
***** 3. row *****
      id:      1
  select_type: SIMPLE
        table: actor
         type:  eq_ref
possible_keys: PRIMARY
         key:   PRIMARY
        key_len: 2
         ref:   sakila.film_actor.actor_id
         rows:  1
       Extra:

```

这说明了 MySQL 为什么要颠倒联接顺序，因为这样可以减少从第 1 个表中读取的行数（注 11）。在这两个例子中，执行引擎可以对第 2 个和第 3 个表进行快速索引查找。区别是它们会进行多少次索引查找：

- 将 film 表放在第一位，对 film 表的每一行都会读取 951 次 film_actor 表和 actor 表。

注 11：严格说来，MySQL 不会尝试减少读取的行数，它只会试着优化对页面的读取。但是行数可以大致显示查询的开销。

- 如果服务器首先扫描 actor 表, 对后续的表就只有 200 次索引查找。

换句话说, 颠倒联接顺序会减少回溯和重新读取。为了确认优化器的选择, 实际执行这两个查询, 然后查看 Last_query_cost 变量。联接顺序相反的查询的开销是 241, 而联接顺序不变的查询的开销是 1154 次。

这只是一个简单的例子, 用于说明 MySQL 联接优化器可以通过对查询重新排序, 以得到较小的开销。对联接重新排序通常是一种非常有效的优化手段。重新排序有时不会得到优化的方案, 这时就可以使用 STRAIGHT_JOIN 参数, 并且按照你认为最佳的方式来组织联接的顺序, 但是这种情况是很少见的。联接优化器比人更能准确地计算开销。

联接优化器试着产生最低开销的查询计划。在可能的时候, 它会从单表计划开始, 检查所有可能的子树的组合。

不幸的是, 对于 n 个表联接, 那么要检查组合的数量就是 n 的阶乘。这个数量被称为“搜索空间 (Search Space)”, 它增长得非常快。如果一个查询要联接 10 个表, 那么检查的数量将是 3 628 800 种 ($10! = 3\,628\,800$)。当搜索空间非常巨大的时候, 优化耗费的时间就会非常长, 这时服务器就不会执行完整的分析。当表的数量超过 optimizer_search_depth 的值时, 它就会走捷径, 比如执行所谓的“贪婪”搜索。

MySQL 在多年的研究和实验中积累了很多经验, 这通常有助于加速优化过程。这些经验是有好处的, 但也意味着 MySQL 也许会因为没检查所有的查询计划而错过优化方案 (这种情况很少)。

有时查询不能被重新排序。联接优化器会利用这种客观情况, 除掉一些选择, 从而减小“搜索空间”。和关联查询一样, LEFT JOIN 也是一个很好的例子 (稍后有更多和子查询相关的内容)。其原因是某张表的结果依赖于另一张表中取得的数据。这种依赖性有助于联接优化器通过消除组合来减少搜索的空间。

排序优化

对结果进行排序可能会有较大开销, 所以常常可以用不排序或只对较少的行排序来改进性能。

第 3 章阐述了如何利用索引来排序。当 MySQL 不能使用索引时, 它就必须自己对结果进行排序。排序可以在内存或磁盘中进行, 但是这个过程叫文件排序 (Filesort), 尽管它实际没有使用文件。

如果待排序的值的数量和排序缓存的大小相当, MySQL 就可以在内存中利用快速排序对所有数据排序。如果 MySQL 不能在内存中排序, 它就会在磁盘上对数据进行分块, 并且对每一块数据都使用快速排序算法, 然后把所有块的数据合并到结果中。

有两种文件排序方法:

双路排序 (Two Passes) ——表算法

读取行指针和 ORDER BY 列, 对它们进行排序, 然后扫描已经排好序的列表, 按照列表中的值重新从表中读取对应的行进行输出。

双路排序的开销可能会非常巨大, 因为它会读取表两次, 第二次读取引发了大量的随机 I/O。对于 MyISAM 表来说, 这个操作的代价尤其高昂。MyISAM 表利用系统调用去提取每行的数据 (它依赖于系统缓存来存储数据)。在另一方面, 它在排序期间保存了最少的数据, 所以如果被排序的行都在内存里面, 那么它就可以在产生最终结果的时候保存和重新读取较少的数据。

单路排序 (Single Pass) ——新算法

读取查询需要的所有列, 按照 ORDER BY 列对它们进行排序, 然后扫描排序后的列表并且输出特定的列。

这个算法在 MySQL4.1 及其后续版本中可用。它的效率高一些，尤其对于 I/O 密集型的数据集，它避免了对表的二次读取，并且把随机 I/O 变成了顺序 I/O。但是，它会使用更多的空间，因为它把需要的每一行的所有列都保存在了内存中，这意味着更少的元组刚好适合排序缓存，并且文件排序会执行更多的合并工作。

MySQL 可能会使用多于预料的临时存储空间，因为它为每一个排序的元组分配了固定的大小。这些记录可能会足够大，以保存最大的数据元组，包括每一个 VARCHAR 列的最大长度，并且，如果使用 UTF-8，MySQL 将为每一个字符分配 3 个字节。这样某些优化很差的架构会占用比磁盘上表实际大小大得多的临时存储空间。

在排序联接的时候，MySQL 在执行查询的期间可能会分两步来执行文件排序。如果 ORDER BY 子句只引用了联接中的第一个表，MySQL 会先对该表进行排序，然后处理联接。如果是这种情况，那么 EXPLAIN 中的 Extra 列就会显示“使用文件排序 (Using filesort)”。否则，MySQL 必须把结果保存到临时表中，然后对临时表进行排序。在这种情况下，EXPLAIN 在 Extra 列中显示的是“使用临时表、使用文件排序 (Using temporary, Using filesort)”。如果有 LIMIT 子句，它将在排序之后生效，所以临时表和文件排序可能会非常大。

参阅第 300 页“优化 filesort”了解如何为文件排序进行服务器调优，以及如何改变服务器使用的排序算法。

178 4.3.4 查询执行引擎

The Query Execution Engine

解析和优化步骤生成了查询执行计划，MySQL 执行引擎使用它来处理查询。查询计划是一个数据结构，而不是其他数据库用于执行查询的可执行字节码。

和优化部分相反，执行部分通常不会很复杂。MySQL 简单地按照执行计划的指令进行查询。计划中的许多操作都是通过存储引擎提供的方法来完成的，这些方法叫“处理器 API (Handler API)”。查询中的每一个表都用一个处理器的实例代替。例如，一个表在查询中出现了 3 次，那么服务器就会创建 3 个处理器。尽管前文没有提及该内容，但是要知道处理器实例是在优化阶段早期产生的，优化器使用它们来获取一些表的信息，比如列名和索引统计。

存储引擎有许多功能，但是它仅仅需要 12 种或所谓的“积木 (Buildingblock)”操作来执行大部分查询。比如，一个操作读取索引中的第 1 行，另一个操作读取第 2 行，这对于执行索引扫描的查询足够了。这种简单的执行方式让 MySQL 的执行引擎结构变得简易，但是也造成了一些优化器的限制。



提示：并不是所有的操作都是处理器操作，比如服务器管理表锁就不是。处理器可能会执行自己的低层次锁定，就像 InnoDB 处理行锁那样，但是这不会代替服务器自己的锁定实现。第 1 章已经说过，所有存储引擎共享的资源都服务器内部实现，例如日期和时间函数、视图和触发器。

为了执行查询，服务器不停地重复指令，直到没有更多的行需要检查。

4.3.5 返回结果到客户端

Returning Results to the Client

执行查询的最后一步是发送结果到客户端。即使查询没有结果要返回，服务器也会对客户端的联接进行应答，比如有多少行受到了影响。

如果查询是可缓存的，MySQL 会在这时缓存查询。

服务器增量地产生和发送结果。回想一下 MySQL 的执行机制就可以知道，一旦它处理了最后一个表并且成功地产生了一行输出，它就会把这个结果发送到客户端。

这么做有两个好处：一是服务器不用把这一行保存在内存中，二是客户端可以尽快地开始工作。（注 12）

4.4 MySQL 查询优化器的限制

Limitations of the MySQL Query Optimizer

MySQL 中“每个查询都是一个嵌套循环联接”的方式并不是优化所有类型联接的最佳方法。幸运的是，MySQL 查询优化器只对很少的查询不适用，而这些查询都可以改写成更有效的方式。



提示：这部分的信息只适用于写本书时的 MySQL 版本，也就是 MySQL5.1。一些限制在未来的版本中可能会被完全取消掉，还有一些限制已经被修复了。尤其值得一提的是，MySQL6 包含相当数量对子查询的优化，而且还有更多的优化正在进行中。

4.4.1 关联子查询 (Correlated Subqueries)

Correlated Subqueries

MySQL 有时把子查询优化得很差。最差的就是在 WHERE 子句中使用 IN。下面有个例子，它将会从 sakila 数据库的 sakila.film 表中找到所有包含女演员 Penelope Guinness (actor_id=1) 的电影。这自然会想到使用子查询：

```
mysql> SELECT * FROM sakila.film
-> WHERE film_id IN(
->     SELECT film_id FROM sakila.film_actor WHERE actor_id = 1);
```

按照设想，MySQL 会由内向外地执行查询，先发现发现一系列的 film_id（译注 2），然后替换掉 IN 里面的内容。通常 IN 列表都很快，所以我们期望这个查询被优化成下面这样：

```
-- SELECT GROUP_CONCAT(film_id) FROM sakila.film_actor WHERE actor_id = 1;
-- Result: 1,23,25,106,140,166,277,361,438,499,506,509,605,635,749,832,939,970,980
SELECT * FROM sakila.film
WHERE film_id
IN(1,23,25,106,140,166,277,361,438,499,506,509,605,635,749,832,939,970,980);
```

不幸的是，实际情况和这正好相反。MySQL 试着让它和外面的表产生联系来“帮助”优化查询，它认为这样检索行会更有效率。所以查询会被重写成：

```
SELECT * FROM sakila.Film
WHERE EXISTS (
    SELECT * FROM sakila.film_actor WHERE actor_id = 1
    AND film_actor.film_id = film.film_id);
```

注 12：如果需要，这种行为是可以改变的，例如，使用 SQL_BUFFER_RESULT 提示。请参阅本章第 195 页“查询优化提示”。

译注 2：原书是 actor_id，疑有错。

现在子查询从外部 film 表中获取 film_id，它不能被最先执行。EXPLAIN 显示它是 DEPENDENT SUBQUERY。可以用 EXPLAIN EXTEND 查看这个查询是如何被改写的。

```
mysql> EXPLAIN SELECT * FROM sakila.film ...;
+----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys |
+----+-----+-----+-----+-----+
| 1 | PRIMARY | film | ALL | NULL |
| 2 | DEPENDENT SUBQUERY | film_actor | eq_ref | PRIMARY,idx_fk_film_id |
+----+-----+-----+-----+-----+
```

EXPLAIN 的输出表明 MySQL 将会对 film 表做全表扫描，并且对每一行执行一次子查询。对于较小的表，这不会对性能造成显著的影响，但是如果外部表很大，那么性能就会非常差。幸运的是，可以把这个子查询改写为联接的方式：

```
mysql> SELECT film.* FROM sakila.film
-> INNER JOIN sakila.film_actor USING(film_id)
-> WHERE actor_id = 1;
```

另外一种优化方式是手工产生 IN 列表，用 GROUP_CONCAT() 单独执行子查询。有时候这种方式比联接快。

MySQL 因为某些特定类型的子查询的执行方案受到了广泛的批评。尽管这肯定需要修复，但是批评往往混淆了两个不同的问题：执行顺序和缓存。从里向外执行查询是一种优化的方式，缓存内部查询的结果是另外一种方式。自己重写查询可以兼顾这两方面。MySQL 的新版本应该会大幅度地优化这种查询，尽管这不是容易的任务。每一种执行计划都会有最坏的情况，即使是人们认为较容易优化的从内向外的执行计划也不例外。

什么时候选择关联子查询

MySQL 不会总是把关联子查询优化得很差。如果别人告诉你要避免子查询，不要听从这个意见！相反地，应该进行评测并且做出你自己的决定。有时关联查询是一种可以得到结果的、极为合理的，甚至最优的方式。看下面这个例子：

```
mysql> EXPLAIN SELECT film_id, language_id FROM sakila.film
-> WHERE NOT EXISTS(
-> SELECT * FROM sakila.film_actor
-> WHERE film_actor.film_id = film.film_id
-> )\G
***** 1. row *****
id:1
select_type: PRIMARY
table: film
type: ALL
possible_keys: NULL
key: NULL
key_len: NULL
ref: NULL
rows: 951
Extra: Using where
***** 2. row *****
id: 2
select_type: DEPENDENT SUBQUERY
table: film_actor
type: ref
possible_keys: idx_fk_film_id
```

181

```

        key: idx_fk_film_id
key_len: 2
        ref: film.film_id
        rows: 2
Extra: Using where; Using index

```

对于这个查询的标准建议就是写一个左外联接 (LEFT OUTER JOIN),而不是使用子查询。从理论上,MySQL 对这两者生成的执行计划应该是一样的,如下面所示:

```

mysql> EXPLAIN SELECT film.film_id, film.language_id
-> FROM sakila.film
-> LEFT OUTER JOIN sakila.film_actor USING(film_id)
-> WHERE film_actor.film_id IS NULL\G
***** 1. row *****
      id: 1
select_type: SIMPLE
      table: film
      type: ALL
possible_keys: NULL
      key: NULL
      key_len: NULL
      ref: NULL
      rows: 951
Extra:
***** 2. row *****
      id: 1
select_type: SIMPLE
      table: film_actor
      type: ref
possible_keys: idx_fk_film_id
      key: idx_fk_film_id
      key_len: 2
      ref: sakila.film.film_id
      rows: 2
Extra: Using where; Using index; Not exists

```

两个计划基本完全一样,但还是有一些区别:

- 一个查询中 film_actor 的 SELECT 类型是 DEPENDENT SUBQUERY,而另外一个为 SIMPLE。这种区别是查询语法的简单反映,因为第 1 个查询使用了子查询,而第 2 个没有。在实际处理上,它不会造成什么不同。
- 对于 film 表,第 2 个查询在 EXTRA 列没有 “Using Where”。但是这也没什么不同,第 2 个查询的 USING 子句和 WHERE 完全是一样的。
- 第 2 个查询在 film_actor 表的 EXTRA 列的值是 “Not Exists”。这是本章开头提到的早期终结的一个例子。它意味着 MySQL 正在使用 “不存在” 这种优化手段,以避免在 film_actor 表中 idx_fk_film_id 索引上读取过多的列。这和 NOT EXISTS 等价,因为它也是一旦发现匹配就立即停止处理当前的行。

所以,从理论上来说,MySQL 几乎用完全一样的执行计划来运行查询。那么在现实就只能用基准测试来分辨真正地快慢。我们使用标准设置对这两个查询进行了评测,结果列在表 4-1 中。

表 4-1: NOT EXISTS 和 LEFT OUTER JOIN 的对比

| Query | 每秒得到的查询结果 (QPS, Query Result per Second) |
|------------------------|--|
| NOT EXISTS 子查询 | 360 QPS |
| 左外联接 (LEFT OUTER JOIN) | 425 QPS |

测试表明子查询要慢得多。

然而，事情并不总是这样。有时候子查询会快一些，例如查看某个表中与另外一个表的记录相匹配的数据。尽管这听上去完全就是一个联接，但是并非总是如此。下面的联接查询的目的是找到任意一部有男演员的电影，因为有些电影有多个男演员，所以它会返回重复的值：

```
mysql> SELECT film.film_id FROM sakila.film
-> INNER JOIN sakila.film_actor USING(film_id);
```

这时候需要使用 DISTINCT 和 GROUP BY 来消除重复值：

```
mysql> SELECT DISTINCT film.film_id FROM sakila.film
-> INNER JOIN sakila.film_actor USING(film_id);
```

但是这个查询真正想表达的意思是什么呢？查询语句能很好地表达意图么？EXISTS 在逻辑上很好地表达了“有一个匹配”这个概念，它不会产生任何重复的行，也能够避免使用 GROUP BY 和 DISTINCT。这两个操作也许需要临时表。下面用一个子查询来替换这个查询：

```
mysql> SELECT film_id FROM sakila.film
-> WHERE EXISTS(SELECT * FROM sakila.film_actor
-> WHERE film.film_id = film_actor.film_id);
```

再对这两个查询运行测试，看看哪个更快。结果在表 4-2 中

表 4-2: EXISTS 和 INNER JOIN

| Query | 每秒查询结果数 (QPS Query Result per Second) |
|-----------------|---------------------------------------|
| INNER JOIN | 185 QPS |
| EXISTS subquery | 325 QPS |

在这个例子中，子查询比联接快得多。

这个详细的例子有两个意图：对待子查询不能有绝对的态度；应该用测试来证明对查询计划和执行速度的假设。

联合的限制

MySQL 有时不能把 UNION 外的一些条件“下推”到 UNION 的内部，而这些外部条件本来用于限制结果或者产生优化。

如果认为 UNION 内的每一个查询都可以从外部的 LIMIT 子句获益，或者认为对于某一个查询使用 ORDER BY，然后所有和它在一起的查询都可以使用这个条件，那么就错了。这时候应该把条件添加到 UNION 内部的每一个子句上。例如，对两个巨大的表使用 UNION，并且使用 LIMIT 子句得到前 20 条记录，MySQL 会把这两个表都放到临时表中，然后取前 20 条记录，可以通过把 LIMIT 放到每一个子查询上面避免这种情况。

索引合并优化

MySQL5.0 中引入了索引合并算法。它让 MySQL 可以在查询中对一个表使用多个索引。MySQL 的早期版本只能使用一个索引，所以在单个索引不能完全满足 WHERE 子句中的所有限制条件的时候，MySQL 通常会全表扫描。例如，film_actor 表在 film_id 和 actor_id 上都有索引，但是它们对下面这个查询都不是最好的选择：

```
mysql> SELECT film_id, actor_id FROM sakila.film_actor
-> WHERE actor_id = 1 OR film_id = 1;
```

在老版本的 MySQL 中，这个查询会进行全表扫描，除非把它写到一个 Union 中。

```
mysql> SELECT film_id, actor_id FROM sakila.film_actor WHERE actor_id = 1
-> UNION ALL
-> SELECT film_id, actor_id FROM sakila.film_actor WHERE film_id = 1
-> AND actor_id <> 1;
```

但是在 MySQL5.0 及其以后的版本中，查询可以使用这两个索引，对它们同时扫描，并且合并结果。这种算法有 3 种变体，分别是：对 OR 取并集；对 AND 取交集；对 AND 和 OR 的组合取并集。下面的查询使用了两个索引扫描，可以从 EXPLAIN 的 EXTRA 列看到这一点：

```
mysql> EXPLAIN SELECT film_id, actor_id FROM sakila.film_actor
-> WHERE actor_id = 1 OR film_id = 1\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: film_actor
         type: index_merge
possible_keys: PRIMARY,idx_fk_film_id
         key: PRIMARY,idx_fk_film_id
      key_len: 2,2
         ref: NULL
        rows: 29
      Extra: Using union(PRIMARY,idx_fk_film_id); Using where
```

MySQL 可以对复杂的子句使用这种技巧，所以对有些查询来说，有可能在 EXTRA 列看到嵌套操作。这通常能很好地工作，但有时这种算法的缓冲、排序和合并操作使用了大量的 CPU 和内存资源。对于没有足够区分性的索引，并行扫描会返回大量需要合并操作的列，这种情况就更容易发生。优化器在意的只是随机页面读取，它并不考虑这种开销。这时候查询看上去开销较低，但实际比整表扫描还要慢。密集的内存和 CPU 使用也会影响并发的查询，但是在隔离的环境中运行查询的时候就不会观察到这种现象。这是另外一种需要真实性能测试的例子。

如果查询因为优化器的限制而运行得很慢，那么可以通过 IGNORE INDEX 命令禁止一些索引，或者使用老的 UNION 策略。

相等传递

相等传递可能会带来意想不到的开销。例如某个列上有一个很大的 IN 列表，优化器知道它会和另外的表中某些列相等，因为 WHERE、ON 或 USING 子句保证了它们之间必须相等。

优化器通过把相应的列拷贝到相关表里来共享列表。这通常是有用的，因为它让优化器和执行引擎有更多的机会选择执行 IN 操作的时机。但是如果这个列表非常大，它就可能导致较慢的优化和执行。写本书的时候，MySQL 内部没什么办法来弥补这个缺陷，如果它对你是个问题，你只能修改源代码（它对大多数人都不是问题）。

并行执行

MySQL 不能在多个 CPU 上并行执行一个查询。其他一些数据库系统有这项特性，但是 MySQL 没有。我们在这儿要提醒你不要花大量的时间试图搞清楚 MySQL 如何并行执行查询。

哈希联接 (Hash Join)

在写本书的时候，MySQL 还不能真正地执行哈希联接。所有的查询都是以嵌套联接的方式执行的。但是可以用哈希索引模拟哈希联接。如果不能用内存存储引擎，那么也就不得不模拟哈希索引。参阅 103 页“建立自己的哈希索引”。

松散索引扫描 (Loose Index Scan)

MySQL 从来就不支持松散索引扫描，即扫描不连续的索引。MySQL 索引扫描通常都需要一个确定的起点和终点，即使查询只需要其中一些不连续的行，MySQL 也会扫描起点到终点范围内的所有行。

下面这个例子有助于说明该问题，假设某个表在列 (a,b) 上有索引，要执行的查询是：

```
mysql> SELECT ... FROM tbl WHERE b BETWEEN 2 AND 3;
```

因为索引从 a 开始，但是 WHERE 子句中没有列 a，MySQL 将会全表扫描并且去掉不匹配的行。如图 4-5 所示。

很显然可以用较快的方式来执行该查询。索引的结构（但不是 MySQL 的存储引擎 API）让你可以查找到范围的开始，然后扫描直到结束，再然后进行回溯，跳到下一个范围进行扫描。如图 4-6 所示。

注意 WHERE 子句中没有列 a，这是不需要的，因为它仅仅让查询可以跳过不需要的列（再强调一次，MySQL 现在还不能这么做）。

这肯定是一个很简单的例子，可以简单地通过添加一个不同的索引来优化这个问题。但是很多时候加索引解决不了问题。下面这个例子就是一个查询在索引的第 1 列上有一个范围条件，而在第 2 个列上有一个相等条件。

186

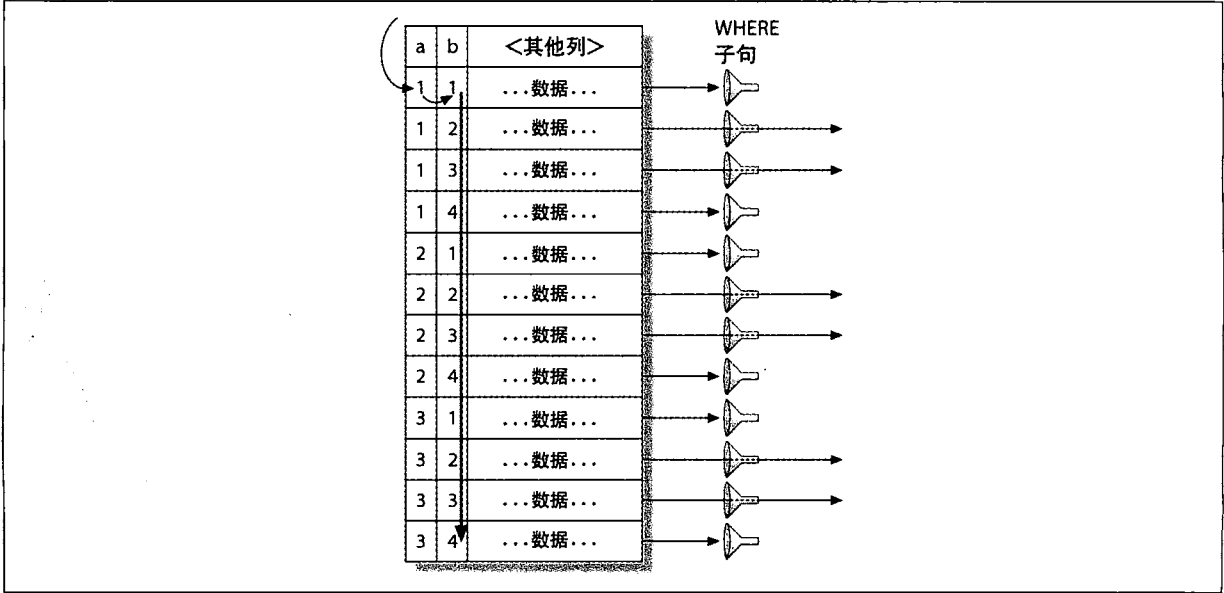


图 4-5：MySQL 通过整表扫描查找数据

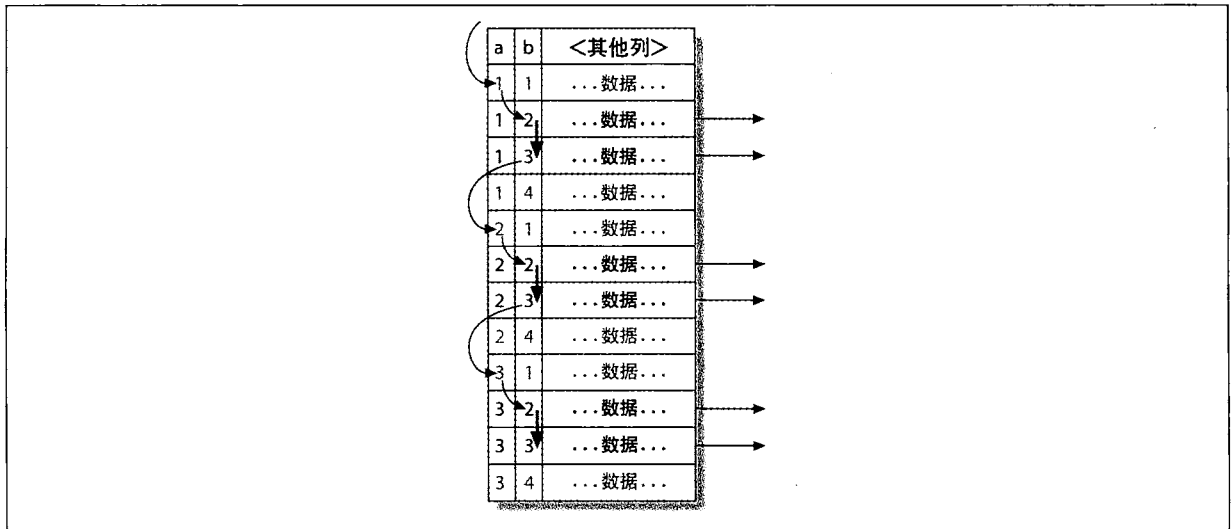


图 4-6: 松散索引扫描。当前的 MySQL 还不支持这种更高效的方式

在 MySQL5.0 中, 可以在某些有限的条件下使用松散索引扫描, 例如在一个分组查询中找最大和最小的值:

```
mysql> EXPLAIN SELECT actor_id, MAX(film_id)
-> FROM sakila.film_actor
-> GROUP BY actor_id\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: film_actor
         type: range
possible_keys: NULL
          key: PRIMARY
       key_len: 2
         ref: NULL
        rows: 396
   Extra: Using index for group-by
```

EXPLAIN 中 “Using Index for group-by” 显示查询使用了松散索引扫描。对于这个特定的查询, 这是一种好的优化措施, 但是它并不是通常意义上的松散索引扫描。所以叫它 “松散索引探测” 会更合适一些。

除非 MySQL 支持真正意义上的松散索引扫描, 否则变通方案就是在索引的第一列提供一个常量或一系列的常量。在前面的对索引进行探讨的章节中, 已经有好几个例子来说明如何针对这种情况取得好的性能了。

MIN()和MAX()

MySQL 不能很好地优化 MIN() 和 MAX()。下面是一个例子:

```
mysql> SELECT MIN(actor_id) FROM sakila.actor WHERE first_name = 'PENELOPE';
```

因为在 first_name 上没有索引, 所以查询会扫描整个表。从理论上说, 如果 MySQL 扫描主键, 它应该在发现第一个匹配之后就立即停止, 因为主键是按照升序排列的, 这意味着后续的行会有较大的 actor_id。但是, 在这个例子中, MySQL 会扫描整个表, 可以通过分析 (Profiling) 查询证实这个结论。一个变通的方式就是去掉 MIN(), 并且使用 LIMIT 来改写这个查询, 就像下面这样:


```
mysql> SELECT actor_id FROM sakila.actor USE INDEX(PRIMARY)
-> WHERE first_name = 'PENELOPE' LIMIT 1;
```

这个通用策略在 MySQL 试图扫描超过需要的行时能很好地工作。如果你是完美主义者，你可能认为这个查询背离了 SQL 的宗旨。我们应该明确地告诉服务器需要什么东西，而服务器也应该知道如何取得数据，因此，在这个例子中，我们告诉 MySQL 如何执行查询，那么与之对应的就是 MySQL 从查询中无法得知我们的真正目的是想得到一个最小值。确实是这样，但是有时候为了高性能，我们不得不对原则进行一些让步。

对同一个表进行 SELECT 和 UPDATE

MySQL 不会让你在对一个表进行 UPDATE 的同时运行 SELECT。实际上这不是优化器的限制。

188 但是知道 MySQL 如何执行查询可以让你找到变通方案。下面是一个展示错误的例子，即使它是一个标准的 SQL 语句。这个查询的目的是找出表中相似行的数量，然后将数量更新到每一行：

```
mysql> UPDATE tbl AS outer_tbl
-> SET cnt = (
-> SELECT count(*) FROM tbl AS inner_tbl
-> WHERE inner_tbl.type = outer_tbl.type
-> );
ERROR 1093 (HY000): You can't specify target table 'outer_tbl' for update in FROM clause
```

一种变通的方式是衍生表。MySQL 把它当成临时表来处理。这样可以有效地处理两个查询，一是在子查询内部使用 SELECT，二是使用表和子查询的结果进行联接，然后进行更新。子查询在外部 UPDATE 打开表之前完成对表的操作，所以查询就可以成功：

```
mysql> UPDATE tbl
-> INNER JOIN(
-> SELECT type, count(*) AS cnt
-> FROM tbl
-> GROUP BY type
-> ) AS der USING(type)
-> SET tbl.cnt = der.cnt;
```

4.5 优化特定类型的查询

Optimizing Specific Types of Queries

本节对如何优化特定类型的查询给出了建议。本书的其他部分已经详细讨论了该主题的大部分内容，但是为了方便查阅，我们把这些内容归纳成了下面的内容。

本节的大部分建议都和版本有关。对未来版本的 MySQL，它们未必适用。所以说，如果某天发现服务器不能应用下面的某些或全部优化，也不用感到迷惑。

4.5.1 优化 COUNT

Optimizing COUNT() Queries

对聚合函数 COUNT 及如何优化使用 COUNT 的查询的误解，估计可以排到“MySQL 十大被误解主题”的头名。在网络上搜索一下，就会发现关于它的误导性信息远比我们想象的要多。在进入优化之前，要明白 COUNT 的真正含义。

COUNT 会干什么

COUNT 是一个特殊的函数，它有两种不同的工作方式：统计值的数量和统计行的数量。值是一个非空 (Non-NULL) 的表达式 (NULL 意味着没有值)。如果在 COUNT () 的括号中定义了列名或其他表达式，COUNT 就会统计这个表达式有值的次数。很多人对这感到费解，部分原因是因为他们弄不清楚值和 NULL 的区别。如果想知道这两者在 SQL 中的含义，我们建议你阅读一本好的入门书籍。互联网不适合获得这个主题的精确信息。

COUNT 的另外一种形式就是统计结果中行的数量。当 MySQL 知道括号中的表达式永远都不会为 NULL 的时候，它就会按这种方式工作。最明显的例子就是 COUNT (*)，它是 COUNT 的一种特例，正如我们所想的那样，它不会把通配符*展开成所有的列，而是忽略所有的列并统计行数。

一个最常见的错误就是在想统计行数的时候，在 COUNT 的括号中放入列名。如果想知道结果的行数，应该总是使用 COUNT (*)，这可以清晰地说明意图，并且得到好的性能。

MyISAM 的迷思

一个通常的误解就是 MyISAM 对于 COUNT 查询非常快。它确实很快，但是只限于很特殊的情况，那就是没有 WHERE 子句的 COUNT (*)，它仅仅是统计表中行的数量而已。MySQL 可以把这种情况优化掉，因为存储引擎总是知道表中行的数量。如果 MySQL 知道某列 (col) 不可能为 NULL，那么它在内部也能把 COUNT (col) 转换为 COUNT (*)。

MyISAM 在查询有 WHERE 子句，或者统计值的时候，不会有魔幻般的优化速度。对于特定的查询，它可能比别的存储引擎快，但也可能慢。这依赖于很多因素。

简单优化

有时可以利用 MyISAM 对 COUNT (*) 的优化对已经有索引的一小部分行做统计。下面的例子使用标准的 world 数据库，展示了如何有效地查找 ID 大于 5 的城市。可以把查询写成下面的形式：

```
mysql> SELECT COUNT(*) FROM world.City WHERE ID > 5;
```

如果使用 SHOW STATUS 分析这个查询，就会发现它扫描了 4 079 行。如果采用否定条件，从总数中减去 ID 小于 5 的城市，就可以把查询改写成下面这样：

```
mysql> SELECT (SELECT COUNT(*) FROM world.City) - COUNT(*)
-> FROM world.City WHERE ID <= 5;
```

这个查询读取的行更少，因为子查询在优化的时候变成了一个常量。在 EXPLAIN 中可以看到：

| id | select_type | table | ... | rows | Extra |
|----|-------------|-------|-----|------|------------------------------|
| 1 | PRIMARY | City | ... | 6 | Using where; Using index |
| 2 | SUBQUERY | NULL | ... | NULL | Select tables optimized away |

有个问题经常在邮件列表和网络论坛里出现，那就是只使用一个查询统计同一列中不同值的数量，以减少查询的数量。例如，使用一个查询统计每种颜色的物品的具体数量。这不能使用 OR，比如 (SELECT COUNT (color='bule' OR color = 'red') FROM items)，这样不能区分每种颜色的数量。也不能把颜色放在 WHERE 子句中，比如 (SELECT COUNT (*) FROM items WHERE color= 'blue' AND color= 'red')，因为颜色

之间是互斥的。下面的查询可以解决这个问题：

```
mysql> SELECT SUM(IF(color = 'blue', 1, 0)) AS blue,  
SUM(IF(color = 'red', 1, 0)) -> AS red FROM items;
```

下面是另外一个等价的查询，但是没有 SUM，而使用了 COUNT，并且保证了如果没有相应的颜色，表达式的值就为 NULL。

```
mysql> SELECT COUNT(color = 'blue' OR NULL) AS blue, COUNT(color = 'red' OR NULL)  
-> AS red FROM items;
```

更多复杂的优化

通常说来，使用了 COUNT 的查询很难优化，因为它们通常需要统计很多行（访问很多数据）。在 MySQL 内部优化它的唯一其他选择就是使用覆盖索引（参阅第 3 章）。如果这还不够，那么就需要更改应用程序架构。可以考虑使用汇总表 (Summary Table) (参阅第 3 章)，还可以利用外部缓存系统，比如数据库缓存服务器 (MemCached)。在优化过程中，通常都会面临相似的窘境，那就是只能在快速、精确、简单 3 个特性中选择两个。

4.5.2 优化联接

Optimizing JOINs

关于这部分的讨论实际贯穿全书，但是这儿有一些值得强调的条目：

- 确保 ON 或 USING 使用的列上有索引（参阅第 95 页“索引基础知识”）。在添加索引时要考虑联接的顺序。比如联接表 A 和 B 的时候使用了列 c，并且优化器按照从 B 到 A 的顺序联接，就不需要在表 B 上添加索引。没有使用的索引会带来额外的开销。通常说来，只需在联接中的第 2 个表上添加索引，除非因为其他的原因需要在第 1 个表上添加索引。
- 确保 GROUP BY 或 ORDER BY 只引用一个表中的列，这样，MySQL 可以尝试对这些操作使用索引。
- 要谨慎地升级 MySQL。因为在不同的版本中，联接的语法、运算符的优先级及其他行为会发生改变。过去曾经发生过一些意外的情况，比如一个普通的联接变成了一个叉积，原本等价的联接返回不同的值，甚至出现语法错误。

4.5.3 优化子查询

Optimizing Subqueries

对子查询最重要的建议就是尽可能地使用联接，至少在当前版本的 MySQL 中是这样。前文已经对这个问题做了广泛的探讨。

子查询是优化器设计小组着力改进的部分，即将发布的 MySQL 版本也许会有更多的子查询优化。但是哪些优化会被最终发布，它们会造成多大的改变还未确定。现在的“联接优先”建议并不适用于以后的版本。服务器正在变得越来越智能，需要你告诉它如何做事情，而不是告诉它返回什么结果的时候会越来越少。

4.5.4 优化 GROUP BY 和 DISTINCT

在很多情况下，MySQL 对这两种方式的优化基本都是一样的。实际上，优化过程要求它们可以互相转化。这两种查询都可以从索引受益，通常说来，索引也是优化它们的最重要手段。

当不能使用索引时，MySQL 有两种优化 GROUP BY 的策略：使用临时表或文件排序进行分组。任何一种方式对于特定的查询都有可能是高效的。可以使用 SQL_SMALL_RESULT 强制 MySQL 选择临时表，或者使用 SQL_BIG_RESULT 强制它使用文件排序。

如果要对联接进行分组，那么通常对表的 ID 列进行分组会更加高效，例如下面的查询效率就不够高：

```
mysql> SELECT actor.first_name, actor.last_name, COUNT(*)
-> FROM sakila.film_actor
-> INNER JOIN sakila.actor USING(actor_id)
-> GROUP BY actor.first_name, actor.last_name;
```

而下面的查询效率会更高：

```
mysql> SELECT actor.first_name, actor.last_name, COUNT(*)
-> FROM sakila.film_actor
-> INNER JOIN sakila.actor USING(actor_id)
-> GROUP BY film_actor.actor_id;
```

按照 actor.actor_id 分组比 film_actor.actor_id 效率更高。可以通过对数据的分析或评测来证实这一点。

这个查询利用了演员的姓名依赖于 actor_id 这一事实，所以它会返回同样的结果。但这并不意味着每次在 SELECT 中选择非分组的列都会得到同样的结果，可以通过配置 SQL_MODE 参数来禁止 SELECT 中使用未在 GROUP BY 中出现的列。如果根本不在意得到的值，或者知道每个分组中的数据都是不同的，那么就可以使用 MIN() 或 MAX() 绕过 SQL_MODE 的限制。就像下面这样：

```
mysql> SELECT MIN(actor.first_name), MAX(actor.last_name), ...;
```

完美主义者会认为分组条件用错了，这种看法是正确的。虚假的 MIN() 或 MAX() 说明查询的结构有问题，但是有时候我们只想让 MySQL 尽可能快地执行查询。完美主义者会喜欢下面这个查询方案：

```
mysql> SELECT actor.first_name, actor.last_name, c.cnt
-> FROM sakila.actor
-> INNER JOIN (
-> SELECT actor_id, COUNT(*) AS cnt
-> FROM sakila.film_actor
-> GROUP BY actor_id
-> ) AS c USING(actor_id)
```

子查询会创建并填充临时表，有时这种方式带来的开销会比稍微变通一点的方案要高一些。要记住，子查询创建的临时表是没有索引的。

在一个分组查询中，SELECT 子句使用非分组的列通常都不是一个好主意，因为结果可能是不确定的，并且如果更改了索引或优化器采用了不同的策略，那么结果也可能被轻易地改变。大部分这样的查询都应该看成“事故”（服务器不会对这种查询发出警告信息），它们也可能是懒惰的结果，但是这肯定不是为了优化而故意设计的。最好可以显式地报告这种情况。我们建议在服务器的 SQL_MODE 参数中加上 ONLY_FULL_GROUP_BY，这样服务器就会对这种查询产生一个错误信息。

除非定义了 ORDER BY，否则 MySQL 会自动对 GROUP BY 里面的列进行排序。如果不在意数据的顺序，可以使用 ORDER BY NULL 来跳过自动排序。也可以在 GROUP BY 后面加上 ASC（升序）或者 DESC（降序）来限定排序的类别。

193 使用 ROLLUP 优化 GROUP BY

分组查询的一个变化就是要求 MySQL 在结果内部实现超级聚合（Super Aggregation）。可以在 GROUP BY 后面加上 WITH ROLLUP 来实现这个需求，但是它也许没有被很好地优化。可以使用解释器检查执行方法，确认分组是否已经通过文件排序或临时表完成，然后试着移除 WITH ROLLUP，并且查看分组方法是否没有变化。可以通过前文中的办法来强制选择分组方法。

有时在应用程序里面进行超级聚合（Super Aggregation）会更好，尽管那意味着要从服务器提取更多列。也可以在 FROM 子句中使用子查询或临时表来保持中间结果。

最好的方式是把 WITH ROLLUP 移到应用程序里面。

4.5.5 优化 LIMIT 和 OFFSET

Optimizing LIMIT and OFFSET

在分页系统中使用 LIMIT 和 OFFSET 是很常见的，它们通常也会和 ORDER BY 一起使用。索引对排序较有帮助，如果没有索引就需要大量文件排序。

一个常见的问题是偏移量很大，比如查询使用了 LIMIT 10000, 20，它就会产生 10 020 行数据，并且丢掉前 10 000 行。这个操作的代价非常高。假设所有页面的访问频率相等，平均每个查询扫描表的一半数据，为了这种查询，可以限制一个分页里访问的页面数目，或者让偏移量很大时查询效率更高。

一个提高效率的简单技巧就是在覆盖索引上进行偏移，而不是对全行数据进行偏移。可以将从覆盖索引上提取出来的数据和全行数据进行联接，然后取得需要的列。这会更有效率。看看下面的查询：

```
mysql> SELECT film_id, description FROM sakila.film ORDER BY title LIMIT 50, 5;
```

如果表非常大，这个查询最好写成下面的样子：

```
mysql> SELECT film.film_id, film.description
-> FROM sakila.film
->   INNER JOIN (
->     SELECT film_id FROM sakila.film
->     ORDER BY title LIMIT 50, 5
->   ) AS lim USING(film_id);
```

这种方式效率更高，它让服务器在索引上面检查尽可能少的数据，一旦取得了需要的行，就把它们联接到完整的表上面，并取出其余的列。相似的技巧可以应用到有 LIMIT 子句的联接上面。

194 有时可以把 LIMIT 转换为位置性查询，服务器可以以索引范围扫描的方式来执行。例如，如果预先计算并且索引了一个表示位置的列，那么就可以把查询写成下面这样：

```
mysql> SELECT film_id, description FROM sakila.film
->   WHERE position BETWEEN 50 AND 54 ORDER BY position;
```

类似的问题还有对数据进行排名，但它往往和 GROUP BY 混合在一起，基本可以肯定的是需要预先计算和存储排名。

如果确实需要优化分页系统，也许应该利用预先计算好的汇总数据。作为替代方案，可以联接只含有 ORDER BY 子句需要的主键和列的冗余表，也可以使用附录 C 中的 Sphinx。

4.5.6 优化 SQL_CALC_FOUND_ROWS

Optimizing SQL_CALC_FOUND_ROWS

对于分页显示，另外一种常用的技巧就是对含有 LIMIT 的查询添加 SQL_CALC_FOUND_ROWS，这样就可以知道没有 LIMIT 的时候会返回多少行数据。这听上去好像是某种“魔法”，因为服务器预测了将会发现多少行数据。但不幸的是，服务器并不能真正地做到这一点。这个选项只是告诉服务器生成结果并丢掉结果中不需要的部分，而不是在得到需要的数据后就立即停止。这个选项的代价很高。

一个较好的设计就是把页面调度放到“下一页”链接上。假设每页有 20 个结果，那么查询就应该 LIMIT 21 行数据，并且只显示 20 行。如果结果中有第 21 行，就会有下一页。

另外一种办法就是提取并缓存大量的数据，比如 1 000 行数据，然后从缓存中获取后续页面的数据。这种策略让应用程序知道一共有多少数据。如果结果少于 1 000 行，那么应用程序就知道有多少页；如果多于 1 000 行，程序就可以显示“找到的结果多于 1 000 个”。这两种策略的效率都比重复产生完整的结果，然后丢弃绝大部分要高得多。

即使不能使用这两种策略，但可以使用覆盖索引，那么使用单独的 COUNT(*) 也比 SQL_CALC_FOUND_ROWS 快得多。

4.5.7 优化联合

Optimizing UNION

MySQL 总是通过创建并填充临时表的方式执行 UNION，它不能对 UNION 进行太多的优化。

可能需要把 WHERE、LIMIT、ORDER BY 或其他条件手工地（比如将它们恰当地从外部查询拷贝到 UNION 的每个 SELECT 语句中）“下推”到 UNION 中，以帮助优化器优化它。

重要的是始终要使用 UNION ALL，除非需要服务器消除重复的行。如果忽略了 ALL 关键字，MySQL 就会向临时表添加 distinct 选项，它会利用所有行来决定数据的唯一性。这种操作开销很大。但是要知道 ALL 不会删除临时表，MySQL 总是把结果放在临时表中，然后再把它们读取出来，即使在没有必要这么做（比如可以把数据直接返回给客户端）时也会如此。

4.6 查询优化提示

Query Optimizer Hints

如果不满意 MySQL 优化器选择的优化方案，可以使用一些优化提示来控制优化器的行为。下面列出了一些提示，以及使用它们的好时机。可以将适当的提示放入待修改的查询中，它只会影响当前查询。对于每个提示的确切语法，请检查 MySQL 的手册。其中有些提示是和版本相关的，它们是：

这两个提示决定了访问同一个表的语句相对于其他语句的优先级。

HIGH_PRIORITY 告诉 MySQL 将一个 SELECT 语句放在其他语句的前面,以便它修改数据。实际上,MySQL 将它放在队列的前面,而不是在队列中等待。也可以把它用于 INSERT 语句,这时它会取消服务器全局 LOW_PRIORITY 设定。

LOW_PRIORITY 正好相反。如果有其他语句需要访问数据,它就把当前语句放到队列的最后。它就像一个人很有礼貌地把住了旅馆的大门,只要有人等着,他自己就不会进门。可以将这个选项用于 SELECT、INSERT、UPDATE、REPLACE 和 DELETE。

这两个选项在有表锁的存储过程中有效,但是在 InnoDB 或其他有细粒度锁定,或者并发控制的存储引擎上无效。在 MyISAM 上要小心地使用它,因为它们会禁止并发插入,并且极大地降低性能。

HIGH_PRIORITY 和 LOW_PRIORITY 经常引起误解。它们并不是指在查询上分配较多或较少的资源,让查询工作得更好或不好。它们只影响服务器对访问表的队列的处理。

156 DELAYED

这个提示用于 INSERT 和 UPDATE。应用了这个提示的语句会立即返回并且将待插入的列放入缓冲区中,在表空闲的时候再执行插入。它对于记录日志很有用,对于某些需要插入大量数据,对每一个语句都引发 I/O 操作但是又不希望客户等待的应用程序也很有用。它有很多限制,比如,延迟插入不能运行于所有的存储引擎上,并且它也无法使用 LAST_INSERT_ID()。

STRAIGHT_JOIN

这个提示可以用于 SELECT 语句中 SELECT 关键字的后面,也可以用于联接语句。它的第 1 个用途是强制 MySQL 按照查询中表出现的顺序来联接表,第 2 个用途是当它出现在两个联接的表中间时,强制这两个表按照顺序联接。

STRAIGHT_JOIN 在 MySQL 没有选择好的联接顺序,或者当优化器花费很长时间确定联接顺序的时候很有用。在后一种情况下,线程将会在“统计 (Statistics)”状态停留很长的时间,添加这个提示将会减少优化器的搜索空间。

可以使用 EXPLAIN 查看优化器选择的联接顺序,然后按照顺序重写联接,并且加上 STRAIGHT_JOIN 提示。对于某些 WHERE 子句,如果认为固定顺序不会对性能有坏的影响,采用这种方式是好办法。但是,在升级 MySQL 之后最好重新检查一下这些联接,因为新的优化措施可能会因为这个提示而失效。

SQL_SMALL_RESULT 和 SQL_BIG_RESULT

这两个提示适用于 SELECT 语句。它们告诉 MySQL 在 GROUP BY 或 DISTINCT 查询中如何并且何时使用临时表。SQL_SMALL_RESULT 告诉优化器结果集会比较小,可以放在索引过的临时表中,以避免对分组后的数据排序。SQL_BIG_RESULT 的意思就是结果集很大,最好使用磁盘上的临时表进行排序。

SQL_BUFFER_RESULT

这个提示告诉优化器将结果放在临时表中,并且尽快释放掉表锁。这和第 161 页“MySQL 客户端/服务器协议”描述的客户端缓冲不同。当客户端没有使用缓冲的时候,服务器端的缓冲就会派上用场,因为这让

客户端避免消耗大量的内存，并且能及时释放掉锁。这其实是一种交换，使用服务器的内存来代替客户端的内存。

SQL_CACHE 和 SQL_NO_CACHE

SQL_CACHE 表明查询已经存在于缓存中。SQL_NO_CACHE 的意思正好相反。第 5 章将详细描述如何使用它们。

SQL_CALC_FOUND_ROWS

这个提示告诉 MySQL 在有 LIMIT 子句的时候计算完整的结果集。即使只返回 LIMIT 限定的行也是如此。可以通过 FOUND_ROWS() 来取得它得到的行数（参见第 194 页“优化 SQL_CALC_FOUND_ROWS”，了解为什么不要使用这个提示）。

FOR UPDATE 和 LOCK IN SHARE MODE

SELECT 语句使用这两个提示来控制锁定，但是只针对有行级锁的存储引擎。如果你想锁定将要更新的行，或者避免锁向上传递并尽可能快地获得独占锁，它们可以帮你预先锁定匹配的行。

INSERT...SELECT 查询不需要这两个提示。因为 MySQL5.0 中要读取的行上默认就有读锁（可以禁止这种行为，但这并不好，具体原因请参阅第 8 章和第 11 章）。MySQL5.1 在某些条件下会放松这种限制。

在写本书的时候，只有 InnoDB 支持这两个提示。现在还不知道未来的其他存储引擎是否支持它们。在 InnoDB 上应用这两个提示时，要明白它们会禁止某些优化，比如覆盖索引。InnoDB 不能在不访问主键的情况下独占地锁定数据行，因为主键存储了数据行的版本信息。

USE INDEX、IGNORE INDEX 和 FORCE INDEX

这些提示告诉优化器从表中寻找行的时候使用或忽略索引（例如，在决定联接顺序的时候）。在 MySQL5.0 和早期版本中，它们不会影响服务器用来排序和分组的索引。在 MySQL5.1 中，它们有两个可选的参数：FOR ORDER BY 或 FOR GROUP BY。

FORCE INDEX 和 USE INDEX 是一样的，但是它告诉优化器，表扫描比起索引来说代价要高得多，即使索引不是非常有效。在认为优化器没有选择正确的索引，或者因为某些原因想使用索引，比如没有 ORDER BY 的时候进行隐含排序，就可以使用它们。可以参阅在第 193 页的“优化 LIMIT 和 OFFSET”的例子，它展示的是如何利用 LIMIT 高效地得到最小值。

在 MySQL5.0 和新版本中，也有一些系统变量会影响优化器。

Optimizer_search_depth

这个变量告诉优化器检查执行计划的深度。如果查询在“统计 (Statistics)”的状态停留了很长时间，就可以考虑减少这个变量的值。

Optimizer_prune_level

这个变量默认状态下是开启的，它可以让优化器根据检查的行的数量跳过某些查询计划。

这两个提示控制了优化器在决定执行计划时的捷径。捷径对于复杂查询的性能是很重要的，但是它们也会因为效率的因素遗漏掉一些优化的方案。这就是为什么有时需要更改它们的原因。

4.7 用户定义变量

User-Defined Variables

用户定义变量很容易被人遗忘，但它们是设计高效查询的利器。对某些过程化和关系化混合的逻辑，它们尤为有效。纯粹关系化查询将任何数据都当成未排序的集合，并且一次性地操作它们。MySQL 采用了一种更加程序化的方式。这也许是一个弱点，但是如果知道如何利用它，这也是一种优势。用户自定义变量对程序化使用 MySQL 很有帮助。

可以把用户自定义变量看成临时保留值的容器，只要和服务器的连接没有断开，它就是有效的。在定义它们的时候，可以使用 SET 或 SELECT 语句给它们赋值（注 13）。

```
mysql> SET @one := 1;
mysql> SET @min_actor := (SELECT MIN(actor_id) FROM sakila.actor);
mysql> SET @last_week := CURRENT_DATE-INTERVAL 1 WEEK;
```

然后就可以在查询中使用：

```
mysql> SELECT ... WHERE col <= @last_week;
```

在了解它们的长处之前，先看看它们的特殊性和短处，这样就可以知道在什么时候不应该使用它们：

- 它们会禁止查询缓存。
- 不能在需要文字常量或标识的地方使用它们，例如表名、列名和 LIMIT 子句。
- 它们和联接相关，不能在跨联接通信中使用。
- 如果正在使用连接池或持久连接，它们会引起部分代码被隔离，从而无法交互。
- 在 MySQL5.0 以前的版本中，它们是大小写敏感的。所以要注意兼容性问题。
- 不能显式地声明变量类型。决定变量是何种类型的时机在不同的 MySQL 版本中是不同的。最好的方式就是给变量显式地赋一个初始值，以决定其类型。如果需要整型，就赋 0，浮点型则赋 0.0，字符串类型则赋 ''（空字符串）。MySQL 用户自定义变量的类型是动态的，它在赋值的时候才会变化。
- 优化器有时可能会把变量优化掉，造成代码无法按照预定的想法工作。
- 赋值的顺序，甚至赋值的时机，都是不确定的，并且依赖于优化器选择的查询计划。最终结果可能会因此而让人非常困惑。
- :=运算符的优先级低于其他运算符，因此必须仔细地使用括号。
- 未定义的变量不会造成语法错误，所以很容易在不知情的情况下犯错。

变量的一个最重要的特性就是可以在变量赋值的同时使用新赋的值。换句话说，赋值是左值（L-VALUE）运算。下面的例子计算并同时输出了行号：

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, @rownum := @rownum + 1 AS rownum
    -> FROM sakila.actor LIMIT 3;
+-----+-----+
```

注 13：在某些时候也可以使用=来赋值，但是为了避免歧义，最好在所有的地方都使用:=来赋值。

| actor_id | rownum |
|----------|--------|
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |

这并不是一个让人特别感兴趣的例子，因为它只复制了表的主键。但是可以把它用于排名。试着写一个查询，找出 10 名出演电影最多的男演员，另外还需要用一个列表示排名，如果出演的电影数量相同，那么名次也一样。先用一个查询找出男演员和他们出演的电影：

```
mysql> SELECT actor_id, COUNT(*) as cnt
-> FROM sakila.film_actor
-> GROUP BY actor_id
-> ORDER BY cnt DESC
-> LIMIT 10;
```

| actor_id | cnt |
|----------|-----|
| 107 | 42 |
| 102 | 41 |
| 198 | 40 |
| 181 | 39 |
| 23 | 37 |
| 81 | 36 |
| 106 | 35 |
| 60 | 35 |
| 13 | 35 |
| 158 | 35 |

现在加上排名，出演了 35 部电影的演员排名应该相同。可以使用 3 个变量，一个变量保存当前排名，一个变量保存前一个演员出演的电影数量，最后一个变量保存当前演员的电影数量。当电影数量变化的时候，排名也跟着改变，先来试一下：

```
mysql> SET @curr_cnt := 0, @prev_cnt := 0, @rank := 0;
mysql> SELECT actor_id,
-> @curr_cnt := COUNT(*) AS cnt,
-> @rank := IF(@prev_cnt <> @curr_cnt, @rank + 1, @rank) AS rank,
-> @prev_cnt := @curr_cnt AS dummy
-> FROM sakila.film_actor
-> GROUP BY actor_id
-> ORDER BY cnt DESC
-> LIMIT 10;
```

| actor_id | cnt | rank | dummy |
|----------|-----|------|-------|
| 107 | 42 | 0 | 0 |
| 102 | 41 | 0 | 0 |
| ... | | | |

排名居然全部都是零！为什么呢？

没有一个答案适合所有的情况。原因有可能是简单的变量名拼写错误（这个例子没有），也可能是另外的因素。在这个例子中，EXPLAIN 表明生成了一个临时表，并且进行了文件排序，所以真正的原因是变量值计算的时间不同。

这个不可预知的问题在使用自定义变量的时候经常遇到。调试这种问题是很难的，但确实是值得的。比如将演员按照出演的电影数量进行排序，SQL 通常使用二次算法。但用户自定义变量可以达到线性算法，这是较大的进步。

针对这个例子，一个简单的解决办法就是加一个临时表，然后在 FROM 子句中使用子查询：

```
mysql> SET @curr_cnt := 0, @prev_cnt := 0, @rank := 0;
-> SELECT actor_id,
->    @curr_cnt := cnt AS cnt,
->    @rank := IF(@prev_cnt <> @curr_cnt, @rank + 1, @rank) AS rank,
->    @prev_cnt := @curr_cnt AS dummy
-> FROM (
->    SELECT actor_id, COUNT(*) AS cnt
->    FROM sakila.film_actor
->    GROUP BY actor_id
->    ORDER BY cnt DESC
->    LIMIT 10
-> ) as der;
```

| actor_id | cnt | rank | dummy |
|----------|-----|------|-------|
| 107 | 42 | 1 | 42 |
| 102 | 41 | 2 | 41 |
| 198 | 40 | 3 | 40 |
| 181 | 39 | 4 | 39 |
| 23 | 37 | 5 | 37 |
| 81 | 36 | 6 | 36 |
| 106 | 35 | 7 | 35 |
| 60 | 35 | 7 | 35 |
| 13 | 35 | 7 | 35 |
| 158 | 35 | 7 | 35 |

大部分关于自定义变量的问题都是由于在查询的不同阶段对变量进行赋值和读取造成的。比如，在 SELECT 语句中给它们赋值，但是在 WHERE 子句中对它们进行读取，这时它们的值就是不可预测的。下面的查询看上去可以返回一行，但实际却不会。

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, @rownum := @rownum + 1 AS cnt
-> FROM sakila.actor
-> WHERE @rownum <= 1;
```

| actor_id | cnt |
|----------|-----|
| 1 | 1 |
| 2 | 2 |

这是因为 WHERE 和 SELECT 在查询的执行过程中是不同的阶段。如果加入 ORDER BY，问题会变得更明显，因为 ORDER BY 是另外一个阶段。

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, @rownum := @rownum + 1 AS cnt
-> FROM sakila.actor
-> WHERE @rownum <= 1
-> ORDER BY first_name;
```

查询会返回表中的每一行，因为 ORDER BY 加入了文件排序并且 WHERE 子句是在文件排序之前计算的。解决的

办法就是在查询执行的同一阶段对变量进行赋值和读取：

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, @rownum AS rownum
-> FROM sakila.actor
-> WHERE (@rownum := @rownum + 1) <= 1;

+-----+-----+
| actor_id | rownum |
+-----+-----+
|         | 1      |
+-----+-----+
```

小测验：如果把 ORDER BY 加到查询中会发生什么情况？读者可以自己试试看。如果没有得到想要的结果，那么原因是什么？在下面的查询中，ORDER BY 改变了变量的值并且 WHERE 子句对它进行了计算，最终结果会如何？

```
mysql> SET @rownum := 0;
mysql> SELECT actor_id, first_name, @rownum AS rownum
-> FROM sakila.actor
-> WHERE @rownum <= 1
-> ORDER BY first_name, LEAST(0, @rownum := @rownum + 1);
```

如果用户自定义变量的行为完全出乎你的预料，那么可以回到 EXPLAIN，查看 EXTRA 列可能出现的“使用 WHERE (USING WHERE)”、“使用临时表 (USING TEMPORARY)”或“使用文件排序 (USING FILESORT)”。它们会给你答案。

最后一个例子引入了另外一种颇具特色的处理方式：在 LEAST 函数中进行赋值。通过这种方式，变量的值被有效地掩盖了起来，并且不会破坏 ORDER BY 的结果（LEAST 函数总是返回 0）。如果只想对变量进行赋值，并且完全避免副作用，这种方式非常有用。它可以隐藏返回值并且避免访问多余的列，比如上个例子中的 dummy 列。GREATEST()、LENGTH()、ISNULL()、NULLIF()、COALESCE() 和 IF() 这些函数不管是单独使用，还是合起来使用，都能很好地达到这个目的，因为它们都有特殊行为，比如 COALESCE() 在发现参数有值的时候就立即停止执行。

不仅仅是 SELECT 语句，所有的语句中都可以对变量进行赋值。事实上，这是用户自定义变量最好的用处之一。比如可以用它来改写代价很高的查询。将使用子查询计算排名用变量进行改写后，代价就相当于执行一次单个的 UPDATE 语句。

但要得到想要的结果还是有点棘手。有时优化器会将变量当成运行时常量，并且拒绝执行赋值操作，这时将赋值放在 LEAST() 这样的函数中会比较有帮助。另外一个提醒就是在执行前检查变量是否有确定的值。我们有时希望它有值，但是有时希望它没有值。

做一些小实验就能了解自定义变量所有有趣的特性，下面有一些实验方式：

- 计算总数和平均数。
- 对分组查询模拟 FIRST() 和 LAST()。
- 对极大的数进行数学运算。
- 将整个表转换为一个 MD5 哈希值。
- 包装一个样值，当它超过某个边界时，对它进行解包。
- 模拟读写游标。

4.7.1 谨慎地升级 MySQL

Be Careful with MySQL Upgrades

如前文所说，试图比 MySQL 更聪明不是好主意，这通常会造成更多的工作和维护开销，而且得到的益处很少。当升级 MySQL 时，这个问题特别突出，因为已有查询中的优化提示也许会禁用新的优化方案。

MySQL 处理索引的方式不是一成不变的。新版 MySQL 会改变已有索引的使用方式，而且也许还要在新版本中对索引进行调整。例如，MySQL4.0 和更老的版本只能对查询使用的每个表使用一个索引，但是 MySQL5.0 和更新的版本可以使用索引合并。

除了 MySQL 偶尔会对优化器做大的变动之外，每次发布都会包含很多小的改变。这些改变通常影响一些小的行为，例如不用考虑索引的某个条件，而且这些小的改变还会让 MySQL 优化更多的特殊情况。

尽管从理论上说，所有改动都是有益的，但在实际情况中，一些查询在升级后性能反而会降低。如果已经对某个版本使用了较长时间，不管你是否意识到了，你很可能已经对那个版本进行了某些调优。那些优化不一定适合新版本，而且有可能会造成性能降低。

如果在意高性能，那么就应该有适合工作负载的特定基础测试方案。这样就可以在对产品服务器进行升级之前，在开发服务器上做一些评测。同样，在升级之前，应该仔细地阅读新版本的版本说明和已知缺陷的列表。MySQL 的手册以友好的方式提供了一份严重缺陷的列表。

我们在这儿并不是要暗示什么，大部分 MySQL 的升级从总体上带来了更好的性能，但是要谨慎地进行升级。

MySQL 高级特性

MySQL 5.0 和 5.1 引入了许多新特性

MySQL 5.0 和 5.1 引入了许多特性，例如存储过程、视图和触发器。对于使用过其他数据库产品的用户来说，它们都是很熟悉的概念。这些新增特性也吸引了很多 MySQL 的新用户。但是只有在人们大规模地使用这些特性之后，才能知道它们的性能。

本章讨论了这些新增特性及其他的高级主题，包括一些在 MySQL 4.1 或更老版本中就存在的特性。性能是讨论的重点，但是也展示了如何从这些特性中得到最大的益处。

5.1 MySQL 查询缓存

MySQL 5.0 和 5.1 引入了许多新特性

许多数据库系统都可以缓存查询计划，服务器可以根据缓存对相同的查询跳过解析和优化阶段。在某些情况下，MySQL 也能做到这一点。但是它还有一种不同的缓存机制，叫做“查询缓存 (Query Cache)”。这种缓存保存了 SELECT 语句的完整结果集 (Complete Result Set)。本节的主题就是这种缓存。

MySQL 查询缓存保留了查询返回给客户端的完整结果。当缓存命中的时候，服务器马上返回保存的结果，并跳过解析、优化和执行步骤。

查询缓存保留了查询使用过的表，如果表发生了改变，那么缓存就失效了。这种失效的方法比较粗糙，看上去也不够高效，因为某些表的改变不会导致查询结果的改变。但是这种简单方式的开销比较小，而这对于繁忙的系统是很重要的。

查询缓存对应用程序完全透明。程序不用知道 MySQL 是从缓存中返回结果还是通过实际计算返回结果。两种方式返回的结果是一样的。换句话说，查询缓存不会改变语义。不管缓存是打开的还是关闭的，服务器的行为都一样 (注 1)。

205

5.1.1 MySQL 如何检查缓存命中

MySQL 5.0 和 5.1 引入了许多新特性

MySQL 检查缓存命中的方式相当简单快捷。缓存就是一个查找表 (Lookup Table)。查找的键就是查询文本、当前数据库、客户端协议的版本，以及其他少数会影响实际查询结果的因素之哈希值。

注 1：查询缓存实际以一种微妙的方式改变了语义。在默认情况下，一个查询中使用的表即使被 LOCK TABLES 命令锁住了，查询也能被缓存下来。可以通过设置 query_cache_wlock_invalidate 来关闭这个功能。

在检查缓存的时候, MySQL 不会对语句进行解析、正则化或者参数化, 它精确地使用客户端传来的查询语句和其他数据。只要字符大小写、空格或者注释有一点点不同, 查询缓存就认为这是一个不同的查询。这是书写查询语句时要注意的一点。不管怎么说, 使用一致的格式和风格是一个好习惯, 而且在这种情况下还能得到更高的性能。

另外一件值得注意的事情就是查询缓存不会存储有不确定结果的查询。因此, 任何一个包含不确定函数 (比如 `NOW()` 或 `CURRENT_DATE()`) 的查询不会被缓存。同样地, `CURRENT_USER()` 或 `CONNECTION_ID()` 这些由不同用户执行, 将会产生不同的结果的查询也不会被缓存。事实上, 查询缓存不会缓存引用了用户自定义函数、存储函数、用户自定义变量、临时表、mysql 数据库中的表或者任何一个有列级权限的表的查询。请参阅 MySQL 手册了解所有不会被缓存的查询类型。

经常可以听到“如果查询包含不确定函数, MySQL 就不会检查缓存”这样的说法。其实这是不对的。MySQL 只有在解析查询的时候才知道里面是否有不确定函数。缓存查找发生在解析之前。服务器会执行一次不区分大小写的检查来验证查询是否以字母 `SEL` 打头。这就是服务器在进行缓存查找前所做的所有事情。

但是, “如果查询包含 `NOW()` 这样的函数, 服务器就不会在缓存中找到结果”这种说法是正确的。因为即使服务器在早些时候执行了同样的查询, 服务器也不会有缓存的结果。MySQL 一旦发现有阻止缓存的元素存在, 它里面就把查询标记为不可缓存, 并且产生的结果也不会被保持下来。

对于引用了当天日期的查询, 如果想让它被缓存下来, 一个有用的技巧就是用一个字面常量来代替函数, 比如下面的例子:

206

```
... DATE_SUB(CURRENT_DATE, INTERVAL 1 DAY) -- 不可缓存!  
... DATE_SUB('2007-07-14', INTERVAL 1 DAY) -- 可缓存
```

因为查询缓存只针对服务器第一次收到的完整 `SELECT` 语句, 所以查询里面的子查询或视图不能使用缓存, 存储过程中的查询也不能使用缓存。MySQL 5.1 之前的准备语句 (Prepared Statement) 也不能使用缓存。

MySQL 查询缓存可以改善性能, 但是在使用的时候有一些问题值得注意。首先, 开启查询缓存对于读写都增加了某些额外的开销。

- 读取查询在开始之前必须要检查缓存。
- 如果查询是可以被缓存的, 但是不在缓存中, 那么在产生结果之后进行保存会带来一些额外的开销。
- 最后, 写入数据的查询也会有额外的开销, 因为它必须使缓存中相关的数据表失效。

这些开销相对来说较小, 所以查询缓存还是很有好处的。但是, 稍后你会看到, 额外的开销有可能也会增加。

对于 InnoDB 的用户, 另外的问题就是事务限制了查询缓存的失效。当事务内部的语句更改了表, 即使 InnoDB 的多版本机制应当对其他语句隐藏事务的变化, 服务器也会使所有引用了该表的查询缓存失效。直到事务提交之前, 该表会全局地不可缓存。所以不会有任何引用了该表的查询, 不管它是在事务的内部还是外部, 在事务提交之前都能被缓存。因此, 长期运行的事务可以增加查询缓存未命中 (Cache Miss) 的数量。

失效对于大型查询缓存也会是一个问题。如果缓存中有许多查询, 缓存失效就会需要很长的时间并且延缓整个系统的工作。这是因为查询缓存有一个全局锁, 它会阻塞所有访问缓存的查询。在检查查询是否命中, 以及是否有查询失效的时候都会发生访问动作。

5.1.2 缓存如何使用内存

How the Cache Uses Memory

MySQL 将查询缓存完全存储在内存中，所以在对它进行调优之前需要了解它如何使用内存。缓存不仅仅存储了查询结果，它在某种程度上像一个文件系统，它保持了自身的结构，而这些结构有助于它了解哪块内存是空闲的、表和查询之间的映射关系、查询文本、查询结果。

除了用于自身的 40KB 内存，查询缓存的内存池被分为大小可变的块。每一块都知道自己的类型、大小、数据量和指向前一个和后一个逻辑块和物理块的指针。内存块可以分为存储查询结果、查询使用的表的列表、查询文本等类型。然而，不同类型的块的处理方式都一样，所以对查询缓存调优的时候不用区分它们的类型。

服务器启动的时候会初始化查询缓存使用的内存。内存池最开始只有一个块。它的大小是被配置用于缓存的内存大小减去自身需要的 40KB。

在缓存查询结果时，服务器会为查询分配一块空间。如果服务器知道正在缓存一个较大的结果，这个块就会大一些。但是它至少等于 `query_cache_min_res_unit` 的值。不幸的是，服务器不能精确地分配大小，因为分配发生在结果产生之前。服务器不会在内存中生成最终的结果然后发送到客户端，而是每产生一行数据，就发送一行，因为这种方式效率更高。这造成的结果就是：当服务器开始缓存结果的时候，它无法知道结果最终会有多大。

分配内存块的速度相对较慢，因为服务器需要查看可用内存的列表并且找到大小合适的块。因此，服务器会尽量减少分配的次数。当需要缓存结果时，它会创建一个大小至少为最小值的块，并且把结果放入到块中。如果块已经满了，但是还有数据没有保存，服务器就会产生一个新块并且继续存储数据。在保存完成后，如果数据块还有剩余空间，服务器就会裁剪该块，并且把空间并入剩下的空闲空间中。图 5-1 显示了该过程（注 2）。

所谓的服务器“分配块”，并不是意味着向操作系统使用 `malloc()` 这样的函数请求内存空间。`Malloc` 过程只会在创建查询缓存的时候发生一次。这儿的意思是服务器检查数据块的列表并且选择一个最佳的位置放置新块。如果有需要，还会移除最旧的查询以腾出空间。MySQL 服务器管理自己的内存，不依赖于操作系统。

到目前为止，这些过程都非常简单。但是，实际过程比图 5-1 要复杂一些。假设平均结果都很小，服务器同时把结果发送到两个客户端。对数据块的裁剪结果就可能留下比 `query_cache_min_res_unit` 还小的空间，这些空间将来不能用于存储结果。

注 2：出于演示的目的，本节的图形已经进行了简化。服务器实际分配块的过程比这儿介绍的要复杂得多。如果想了解详情，`sql/sql_cache.cc` 这个文件头部的注释将该过程解释得很清楚。

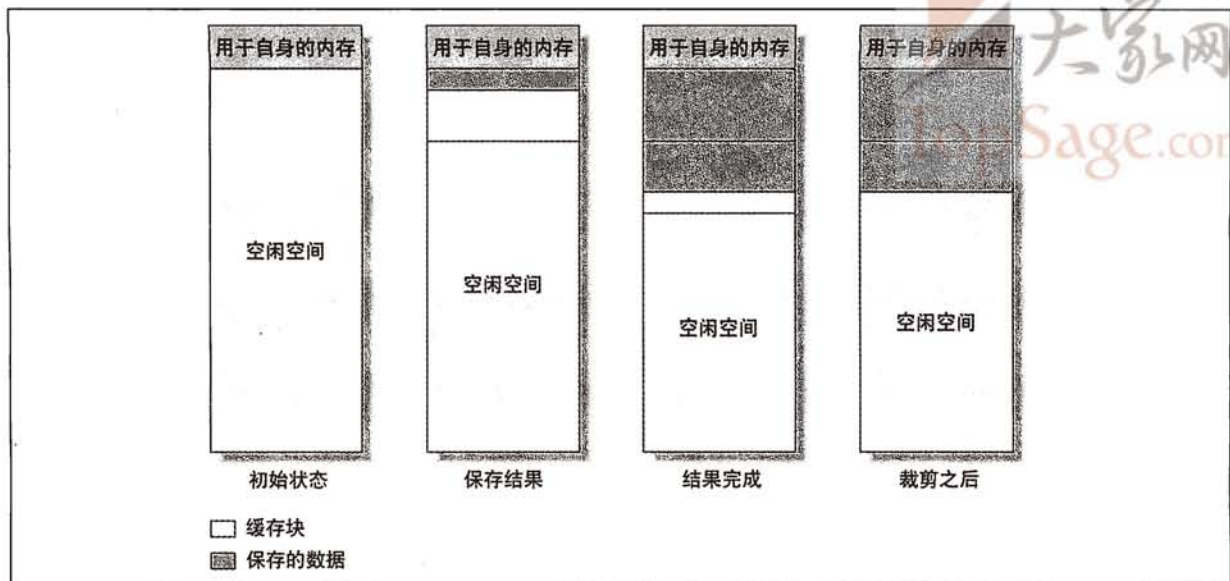


图 5-1：查询缓存如何分配内存块以存储数据

这时，数据块的分配看上去就像图 5-2。

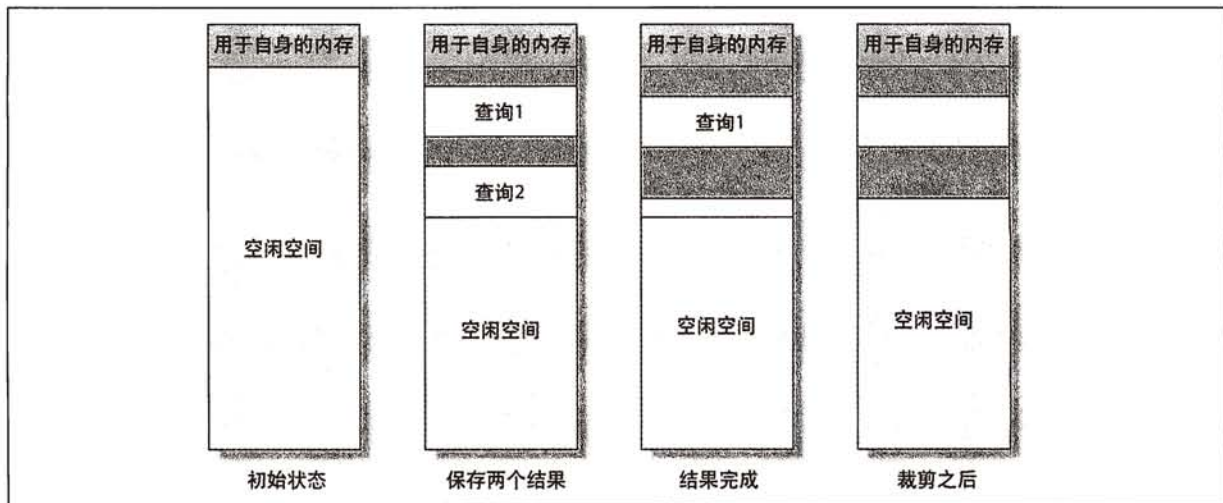


图 5-2：向查询缓存中存储结果引起的碎片

对第一个结果进行裁剪后，在两个结果之间留下了一块很小的空间，它不能存储结果，这就是碎片。它也是内存和文件系统空间分配的经典问题。导致碎片的原因是多样的，比如缓存失效会留下不能复用的小块空间。

5.1.3 查询缓存何时有帮助

When the Query Cache is Helpful

缓存并不会自动地比非缓存高效。缓存也需要开销，只有在节省的资源大于开销的时候，缓存才是真正有效率的，这和服务器的负载相关。

在理论上,可以通过对比在缓存开启和关闭时服务器需要做的工作来了解缓存是否有帮助。在缓存关闭的时候,读取操作需要执行查询语句并且把结果返回给客户,写入操作需要执行查询。在缓存启用的时候,读取首先会检查查询缓存,然后要么直接返回结果,要么执行查询语句,保存结果,再返回结果。每一个写入操作都需要执行查询语句并且检查是否有缓存过的查询要失效。

尽管这听上去很直接,但其实不是。实际上很难精确地计算或者预测查询缓存的好处。必须考虑外部因素。例如,查询缓存可以减少产生结果的时间,但它不会减少将结果发送到客户端的时间,而这有可能是主要因素。

从缓存中受益最多的查询可能是需要很多资源来产生结果,但是不需要很多空间来保存的类型。所以用于存储、返回和失效的代价都较小。聚集查询,比如从大表中利用 `COUNT()` 产生较小的结果,就符合这个范畴。但是也有其他很多种查询值得缓存。

检查是否从查询缓存中受益的最简单的办法就是检查缓存命中率。它是缓存提供的查询结果的数量,而不是服务器执行的数量。当服务器收到 `SELECT` 语句的时候, `Qcache_hits` 和 `Com_select` 这两个变量会根据查询缓存的情况进行递增,查询缓存命中率的计算公式是: $Qcache_hits / (Qcache_hits + Com_select)$ 。

命中率要多少才好?这视情况而定,即使 30% 的命中率也可能很有帮助。因为对于每一个查询,不执行它所节约的资源远大于在缓存中保存结果及让查询失效的开销。知道哪个查询被缓存了也很重要。如果缓存命中代表了开销最大的查询,那么即使是很低的命中率也有很大的好处。

任何一个不在缓存中存在的查询都是**缓存未命中**。缓存未命中可能是因为下面的原因:

- 查询不可缓存。原因可能是含有不确定函数,比如 `CURRENT_DATE`,也有可能是结果太大,无法缓存。状态变量 `Qcache_not_cached` 会因为这两种无法缓存的查询而增加。
- 服务器以前从来没见过这个缓存,所以它根本就没有机会缓存自身结果。
- 查询的结果以前被缓存过,但是服务器把它移除了。发生移除的原因可能是内存空间不够,所以被人从服务器上把它删除了,也可能是缓存失效了。

如果服务器有很多缓存未命中,但是不能缓存的查询却很少,那么原因应该是下面之一:

- 查询缓存未被激活,也就是说服务器根本就没机会将结果存储到缓存中。
- 服务器看到了以前未曾见过的查询。如果没有很多重复的查询,即使缓存被激活了,也有可能见到这种情况。
- 有很多缓存失效。

缓存可能会因为碎片、内存不足或数据改变而失效。如果已经给缓存分配了足够的内存,并且把 `query_cache_min_res_unit` 调整到了合适的值,那么大部分缓存失效都应该是由数据改变引起的。可以通过检查 `Com_*` (`Com_update`, `Com_delete` 等) 的值知道有多少查询修改了数据。也可以通过检查 `Qcache_lowmem_prunes` 的值了解有多少查询因为内存不足而失效。

一个不错的想法就是把缓存失效的开销和命中率分开考虑。举一个极端的例子,假设有一个表只发生读取动作并且命中率是 100%,另外一张表只发生更新。如果只简单地从状态变量计算命中率,它始终都是 100%。然而,查询缓存在这种情况下仍然有可能效率不高,因为更新被减慢了。所有的更新查询在完成后都不得不访问缓存,检查是否有其他的查询会因为数据变化而失效。但实际上答案始终是“否”。所以这些工作就全部浪费了。如果不在检查命中率的同时检查不可缓存的查询数量,那么就不可能发现这些问题。

如果对同一张表进行大致平均的读写，服务器也可能不会从查询缓存中得益。写入数据会不停地让缓存失效，而读取数据会不停地把新结果插入缓存中。在这种情况下，只有发生后续读取，这种缓存才是有益的。

如果在服务器收到同一个查询语句之前，缓存就失效了，那么保持结果就只能是浪费时间和内存。检查 `Com_select` 和 `Qcache_inserts` 的相对大小可以确认这种情况是否发生。如果差不多所有的 `SELECT` 语句都是缓存未命中（`Com_select` 会因此增加），并且接下来把结果保存到了缓存中，那么 `Qcache_inserts` 就会和 `Com_select` 差不多大小。因此，`Qcache_inserts` 一般比 `Com_select` 小得多，至少在缓存被正确地激活后是这样的。

每个应用程序都有确定的潜在缓存大小，即使没有写入查询也是如此。潜在缓存用于保存应用程序所有可缓存查询的内存数量。从理论上说，它对于大部分应用程序是极大的数字。在实际中，由于失效的数量，许多应用程序可用的缓存比期望的小得多。即使把查询缓存设置得非常大，实际也不可能超过潜在缓存的大小。

你应该监视服务器实际使用的缓存数量。如果它没有用到分配的内存，那么就应该把分配给它的内存减少一点。如果由于内存限制引起了缓存失效，那么就应该多分配一些内存。但是不用太在意缓存的大小，它比有实际影响的稍大一点或小一点都没有问题。只有在内存有严重浪费或者缓存失效太多的时候才需要去考虑它的大小。

还应该在服务器其他缓存和查询缓存之间找到某种平衡，例如 InnoDB 的缓存池或 MyISAM 的键缓存。它们之间没有简单的公式或固定的比例，因为这取决于应用程序。

5.1.4 如何对查询缓存进行维护和调优

一旦了解查询缓存的工作机制，对它进行调优就是一件容易的工作。它只有几个“活动部分”。

Query_cache_type

这个选项表示缓存是否被激活。具体选项是 `OFF`、`ON` 或 `DEMAND`。`DEMAND` 的意思是只有包含了 `SQL_CACHE` 选项的查询才能被缓存。它既是会话级变量，也是全局性变量（更多关于会话变量和全局变量的话题请参阅第 6 章）。

Query_cache_size

分配给查询的总内存，以字节为单位。它必须是 1024 的倍数。所以 MySQL 实际使用的值可能和定义的值稍有不同。

Query_cache_min_res_unit

分配缓存块的最小值。第 206 页的“缓存如何使用内存”解释过该设置，下节会对它做进一步讨论。

Query_cache_limit

这个选项限制了 MySQL 存储的最大结果。如果查询的结果比这个值大，那么就不会被缓存。要知道的是服务器在产生结果的同时进行缓存，它无法预先知道结果是否会超过这一限制。如果在缓存的过程中发现已经超过了限制，MySQL 会增加 `Qcache_not_cached` 的值，并且丢掉已经缓存过的值。如果知道会发生这样的事，那么给查询加上 `SQL_NO_CACHE`，可以避免这种开销。

这个选项指是否缓存其他联接已经锁定了的表。默认值是 OFF，可以让你从其他联接已经锁定了的表中读取缓存过的数据，这改变了服务器的语义。因为这种读取通常是不被允许的。把它改成 ON 会阻止读取数据，但有可能增加锁等待。它对于大多数程序都没有影响，所以通常保持默认值就可以了。

在原则上，对缓存进行调优很简单，但是理解自己所做的改变的影响则要复杂得多。接下来的章节展示了如何对查询缓存进行推理，以做出正确的决定。

减少碎片

没有办法避免所有的碎片，但是仔细地选择 `query_cache_min_res_unit` 可以避免在查询缓存中造成大量的内存浪费。关键在于每一个新块和服务已分配给存储结果的块的数量之间找到平衡。如果值过小，服务器将会浪费较少的内存，但会更频繁地分配块，这对服务器意味着更多的工作。如果值过大，碎片将会很多。合适的折中是在浪费内存和增加 CPU 处理时间上取得平衡。

最佳设置根据典型查询结果而定。可以用使用的内存（大致等于 `query_cache_size - Qcache_free_memory`）除以 `Qcache_queries_in_cache` 得到查询的平均大小。如果缓存由大结果和小结果混合而成，那么就很难找到一个合适的大小，既能避免碎片，也能避免过多的内存分配。但是，有理由相信缓存大结果没有太大的益处（这通常是真的）。可以通过降低 `query_cache_limit` 的值阻止缓存大结果，它有时有助于在碎片和在缓存中保存结果的开销中得到平衡。

可以通过检查 `Qcache_free_blocks` 的值来探测缓存中碎片的情况，它可以显示缓存中有多少内存块处于 FREE 状态。图 5-2 中最后一步显示了两个处于 FREE 的块。碎片最严重的情况就是在每两个存储了数据的块之间都有一个比最小值稍小的可用块。这样的话，每隔一个存储块就有一个自由块。因此，如果 `Qcache_free_blocks` 大致等于 `Qcache_total_blocks/2`，则说明碎片非常严重。如果 `Qcache_lowmem_prunes` 的值正在增加，并且大量的自由块，这意味着碎片导致查询正被从缓存中永久删除。

213

可以使用 `FLUSH QUERY CACHE` 命令移除碎片。这个命令会把所有的存储块向上移动，并把自由块移到底部。当它运行的时候，它会阻止访问查询缓存，这锁定了整个服务器。但它通常都很快，除非缓存非常大。和名字相反，它不会从缓存中移除查询，`RESET QUERY CACHE` 才会这么做。

提高查询缓存的可用性

如果缓存没有碎片，但是命中率却不高，那么就应该给缓存分配较少的内存。如果服务器找不到足够大小的块来存储结果，那么就应该从缓存中清理掉一些查询。

当服务器清理查询的时候，`Qcache_lowmem_prunes` 的值会增加。如果它的值增加得很快，那么可能有两个原因：

- 如果有很多自由块，那么问题可能是由碎片引起的（参阅前一节）。
- 如果自由块比较少，那么这可能意味着工作负载使用的内存大小超过了所分配的内存。可以检查 `Qcache_free_memory` 知道未使用的内存数量。

如果有很多自由块，碎片很少，由于内存不足引起的清理工作也很少，但是命中率仍然不高，这说明工作负载也许不能从缓存中受益。肯定有什么东西阻止查询使用缓存，很多 `update` 语句可能是原因，另外一个可能的原因是查询是不可缓存的。

如果已经估算过缓存命中率，但是还不确定服务器是否从缓存中受益，此时可以禁用缓存并且监控性能，然后重新开启缓存并观察性能变化。为了禁用缓存，可以将 `query_cache_size` 设置为 0（改变 `query_cache_type` 不会从全局上影响已经打开了的连接，而且不会把内存归还给服务器）。也可以做基准测试，但是有时候很难得到包含了可缓存的查询、不可缓存的查询，以及更新语句的测试样例。

图 5-3 用一个基本例子显示了分析和调整查询缓存的流程。

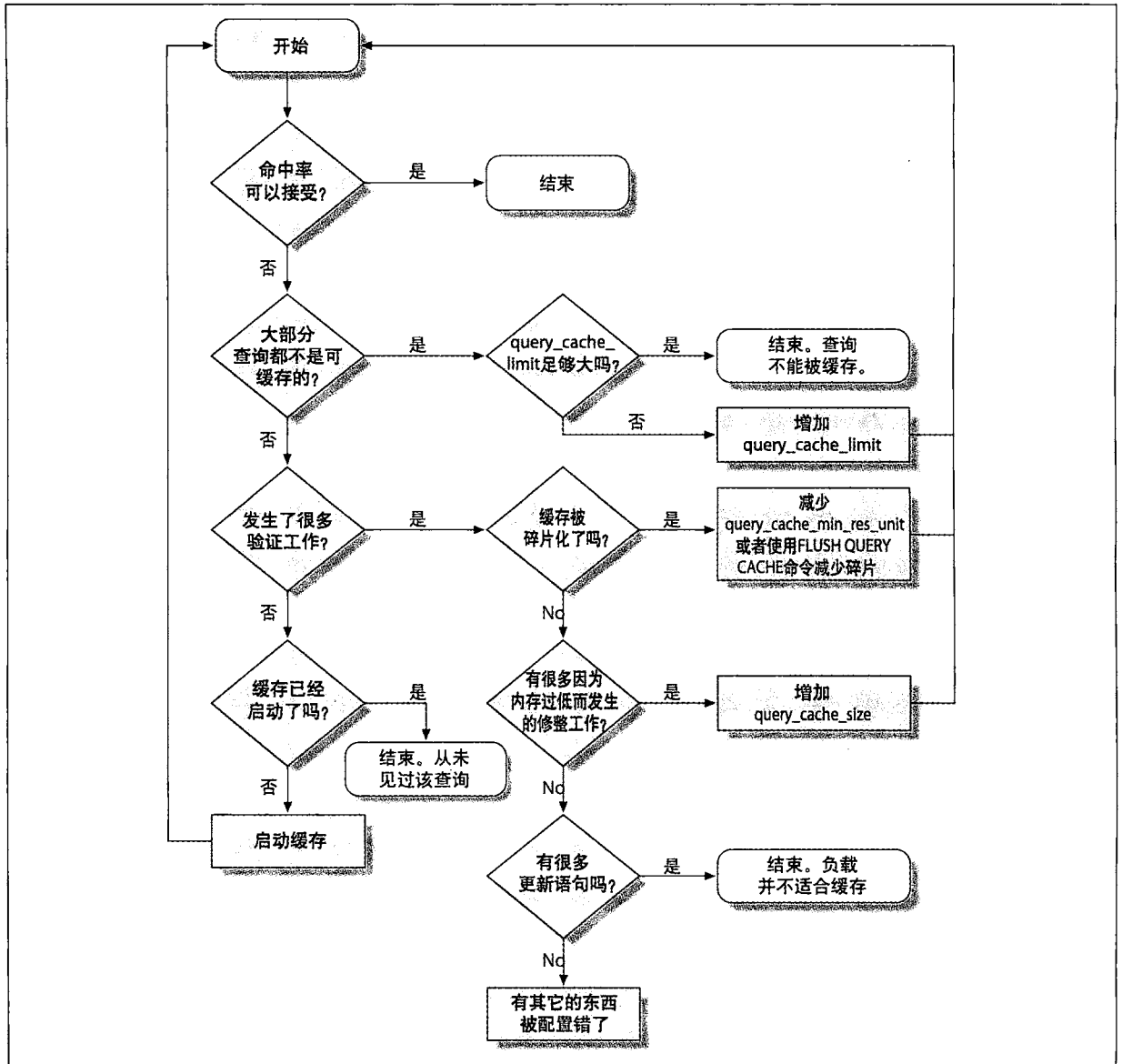


图 5-3：如何分析调整查询缓存

5.1.5 InnoDB 和查询缓存

InnoDB 由于其 MVCC 架构，和查询缓存的交互比其他存储引擎要复杂得多。在 MySQL 4.0 中，查询缓存在事务内部是被完全禁止的，但是在 MySQL 4.1 及其新版本中，InnoDB 会针对每一个表请示服务器一个事务是否可以访问查询缓存。它控制了读（从缓存中获取数据）、写（向缓存中保存数据）操作对缓存的访问。

决定访问的因素是事务 ID 及表上是否有锁。每个表在 InnoDB 的内存内部数据字典中都有一个对应的事务 ID 计数器。ID 小于计数器值的事务会被禁止从缓存读取或写入数据。表上的任何锁都会导致使用该表的查询变得不可缓存。例如，如果事务在表上面执行了 SELECT FOR UPDATE 查询，那么直到锁解除为止，不会有任何其他的事务能从缓存对它进行读写操作。

当事务提交的时候，InnoDB 会在锁定的基础上更新计数器的值。锁可以作为粗略的提示，用来表示事务是否修改了表。有可能事务锁住了表却没有更新它，但是不可能在没有获取锁的情况下修改表的内容。InnoDB 把每个表的计数器设置为系统的事务 ID，它是已有的事务 ID 的最大值。

这会造成下面的结果：

- 使用查询缓存的事务中的表的计数器位于绝对下限。如果系统事务 ID 是 5，并且一个事务请求了表的行锁，然后提交了，事务 1 到 4 中引用的这个表的查询将再也不能对缓存进行读写操作。
- 表的计数器没有按照使用表的事务 ID 进行更新，却按照系统事务 ID 进行了更新。这样的后果就是这个事务发现自己以后再也不能在缓存中操作引用了这个表的查询。

查询缓存的存储、获取及失效都在服务器级进行，并且 InnoDB 不能绕过或延迟这种行为。但是，InnoDB 能显式地告诉服务器让使用了某些特定的表的查询失效。在外键约束，比如 ON DELETE CASCADE，修改了查询中未出现的表的内容时，这是有用的。

从原则上说，如果对表的改动不会影响其他事务看到的连续读取视图，InnoDB 的 MVCC 架构就可以让缓存为查询服务。但是，实现这种方式是很复杂的。InnoDB 的算法出于简单性的考虑走了一些捷径，其代价就是在实际并不需要的时候锁住了查询缓存之外的事务。

5.1.6 通用查询缓存优化方案

架构、查询以及应用程序设计会影响查询缓存。除了上节讨论的内容，下面还有一些需要注意的观点：

- 使用多个较小的表，而不是用一个大表，对查询缓存有帮助。这种设计方式使失效策略工作在一个较好的颗粒度上。但是不要让这个想法过度影响架构设计，因为其他的因素能轻易抵消它的好处。
- 成批地进行写入操作，而不是逐个执行，会有效率得多。因为这种方法只会引起一次失效操作。
- 我们已经注意到在让缓存失效或清理一个大型缓存的时候，服务器可能会挂起相当长时间。至少在 MySQL 5.1 之前的版本中是这样。一个容易的解决办法就是不要让 query_cache_size 太大，256MB 已经太大了。
- 不能在数据库或表的基础上控制查询缓存，但是可以使用 SQL_CACHE 和 SQL_NO_CACHE 决定是否缓存查询。也可以基于某个连接来运行或禁止缓存，可以通过用适当的值设定 query_cache_size 来开启或关闭对某个连接的缓存。

- 对于很多写入任务的应用程序，关闭查询缓存也许能改进性能。这样做可以消除缓存那些很快就会失效的查询所带来的开销。要记住在禁用时需要把 `query_cache_size` 设置到 0，这样就不会消耗任何内存。

如果想让大多数查询都不使用缓存，但是有少部分查询能从缓存中极大地受益，这时可以将全局变量 `query_cache_type` 设置为 `DEMAND`，然后在想使用缓存的查询后面添加 `SQL_CACHE`。尽管这会造成更多的工作，但是可以细粒度地控制缓存。相应地，如果想缓存大部分查询，只排除其中一小部分，就可以使用 `SQL_NO_CACHE`。

5.1.7 替代查询缓存的方法

MySQL 查询缓存的原则就是最快的查询不需要执行，但是仍然需要发起查询，服务器也需要做一点工作。如果某些特殊的查询完全不想和服务端沟通，那该怎么办？客户端缓存可以进一步减轻 MySQL 服务器的负载。第 10 章提供了更多的细节。

217 5.2 在 MySQL 中存储代码

MySQL 可以用触发器、存储过程和存储函数把代码保存在服务器内部。MySQL 5.1 中还可以把代码保存为名为“事件 (Event)”的周期性任务。存储过程和存储函数被统称为“存储例程 (Stored Routines)”。

这 4 种存储代码使用了一种特殊的 SQL 扩展语言，它包括了循环，以及条件判断等过程性结构（注 3）。这些存储代码之间最大的区别就是他们操作的上下文，也就是输入和输出。存储过程和存储函数能接受参数并返回结果，触发器和事件则不能。

从原则上说，存储代码是共享和复用代码的好方式。Giuseppe Maxia 和其他人在 <http://mysql-sr-lib.sourceforge.net> 创建了通用的存储例程库。但是很难在其他的数据库系统中重用代码，因为它们都包含了自己的语言（DB2 是一个例外，它和 MySQL 基于同一个标准，语言很相似）（注 4）。

本书主要集中在存储代码的性能，而不是具体语法上。O'Reilly 公司的《MySQL Stored Procedure Programming》（Guy Harrison 和 Steven Feuerstein 著）对在 MySQL 中书写存储过程很有用。

存储代码的支持者和反对者随处可见。我们不会偏向任意一方，下面列出了它的优点和缺点。首先，它有如下优点：

- 它在代码存在的地方运行，所以可以在服务器内部运行任务，从而节约带宽和减少延迟。
- 它是一种代码复用的方式。它能集中商业逻辑，这可以加强程序行为的一致性并且让人更加省心。
- 它可以减少发布和维护的开销。
- 它提供了安全性方面的优势，以及更好地控制权限的方式。一个通常的例子就是用于转账的存储过程。

注 3：这种语言是 SQL/PSM (Persistent Stored Modules) 的子集。它定义于 ISO/IEC 9075-4:2003(E)。

注 4：一些移植工具，比如 `tsql2mysql` (<http://sourceforge.Net/projects/tsql2mysql>) 可以把 Microsoft SQL Server 转换为 MySQL。

过程可以在事务内部完成转账，并且把所有的操作都记录下来。可以让应用程序调用存储过程而不用授权访问特定的表。

- 服务器缓存了存储过程的执行计划，这可以降低重复调用的开销。
- 存储代码被保存在服务器里面，可以和服务器一起发布、备份和维护。它很适合维护任务。它不依赖于任何外部的组件，比如 Perl 库和其他你不想放在服务器上的软件。
- 它让应用程序程序员和数据库程序员的工作可以分开。数据库专家肯定更适合写存储过程，因为并不是应用程序方面的每一个程序员都善于写高效的查询语句。

它有下面的缺点：

- MySQL 没有提供很好的开发和调试工具，所以在 MySQL 里面写存储代码比在其他数据库服务器要难。
- MySQL 语言比应用程序语言慢而且原始。可用的函数是有限的，并且很难写出复杂的字符串操作和逻辑。
- 存储代码实际增加了部署应用程序的复杂性。除了程序代码和数据库架构的改变，还需要部署服务器内部的代码。
- 因为存储例程和数据库保存在一起，所以它可能变成安全隐患。比如存储例程使用了非标准化的加密函数，那么在数据库受到攻击的时候就无法保护数据。如果加密函数在代码里面，攻击者将危及数据库和代码。
- 存储例程把负载转移到了数据库服务器，它通常更难扩展并且比应用程序和网页服务器更昂贵。
- 你不能通过 MySQL 控制存储代码分配的资源，因此一个错误就能导致服务器宕机。
- MySQL 对存储代码的执行有很多限制，比如执行计划缓存基于单个连接，游标是用临时表实现的，等等（我们在讨论各种特性的时候会涉及限制）。
- 在 MySQL 里面很难剖析存储过程的代码。当缓慢查询显示 `CALL XYZ('A')` 的时候，对它的分析就很难。因为你不得不找到该存储过程并且调查代码。
- 存储代码是一种隐藏复杂性的方式，它简化了开发但是常常没有好的性能。

在质疑是否该使用存储代码的时候，要问问自己业务逻辑会被放在什么地方。是应用程序代码中？还是数据库里面？这两种方式都很普遍，要知道的是，当使用存储代码的时候，业务逻辑就在数据库里面了。

5.2.1 存储过程和函数

MySQL 的架构和查询优化器对使用存储例程（Stored Routines）和它们的效率有一些限制。在写本书时，有如下限制：

- 优化器不能使用存储函数内部的 `DETERMINISTIC` 修饰符把重复调用优化掉。
- 优化器无法估计执行存储函数的开销。
- 每个连接都有自己的存储过程执行计划缓存。如果许多连接调用同一个存储过程，它们将会浪费资源去缓存同样的执行计划（如果使用连接池或持续连接，执行计划将会被缓存得更久一些）。
- 存储例程和复制（Replication）以令人困惑的方式交织在一起。如果不想复制对存储例程的调用，而想复制数据集上发生的实实在在的改变，那么就应该使用 MySQL 5.1 引入的基于行复制的机制。如果在 MySQL 5.0 中开启了二进制日志记录，服务器就会强制要求要么把所有的存储过程定义为

DETERMINISTIC, 要么开启服务器上的 log_bin_trust_function_creators 选项。

最好把存储例程做得既小又简单。最好用面向过程的语言把复杂的逻辑放在数据库外面, 这会更有表达性和通用性。这也可以访问更多的计算资源并且有可能使用不同类型的缓存。

但是, 存储过程对于某些类型的操作是很快的, 尤其是对于小型查询。如果一个查询足够小, 解析和网络通信的开销就会成为主要因素。作为展示, 下面有一个简单的存储过程, 它向一个表插入特定数目的行, 代码如下:

```
1 DROP PROCEDURE IF EXISTS insert_many_rows;
2
3 delimiter //
4
5 CREATE PROCEDURE insert_many_rows (IN loops INT)
6 BEGIN
7     DECLARE v1 INT;
8     SET v1=loops;
9     WHILE v1 > 0 DO
10         INSERT INTO test_table values(NULL,0,
11             'qqqqqqqqqwwwwwwwwweeeeeeeeeerrrrrrrrrrttttttttt',
12             'qqqqqqqqqwwwwwwwwweeeeeeeeeerrrrrrrrrrttttttttt');
13         SET v1 = v1 - 1;
14     END WHILE;
15 END;
16 //
17
18 delimiter ;
```

然后用该存储过程插入 100 万行数据, 并和使用客户端程序每次插入一行相对比。表的结果和使用的硬件没有太大关系, 重要的是它们之间的相对速度。出于好玩的目的, 我们也测试了通过 MySQL 代理连接服务器的时间。为了保持测试的简单性, 在同一台服务器上运行所有的测试。表 5-1 显示了结果。

表 5-1: 一次性插入 100 万行数据耗费的时间

| 方法 | 总时间 |
|--------------------|-------|
| 存储过程 | 101 秒 |
| 客户端程序 | 279 秒 |
| 使用了 MySQL 代理的客户端程序 | 307 秒 |

存储过程要快得多, 主要原因是它避免了网络通信、解析、优化等的开销。

在第 227 页中的“准备语句的 SQL 接口”, 有一个典型的用于维护工作的存储过程示例。

5.2.2 触发器

触发器 (Trigger) 可以在执行 INSERT、UPDATE 或 DELETE 的时候运行代码。触发器可以在这些语句之前或之后运行。触发器不会返回结果, 但是它可以读取或修改数据。因此, 可以使用触发器代替客户端代码来强制约束 (Constraint) 或保证商业逻辑。一个不错的例子就是在类似于 MyISAM 这样的不支持外键的存储引擎上模拟外键。

触发器可以简化程序逻辑并提高效率。因为它节约了数据服务器和客户端之间数据往来的开销。它对自动更新非正则化表和汇总表很有帮助。例如, Sakila 数据库就使用它来维护 film_text 表。

在写作本书的时候，MySQL 触发器还未被完整地实现。如果你习惯于在其他数据库产品中广泛地使用触发器，那么就应该假设它在 MySQL 中不会按照同样的方式工作，尤其是：

- 对于每个事件，在每个表上只能有一个触发器。换句话说，不会有两个触发器启动 AFTER INSERT 事件。
- MySQL 只支持行级触发器，也就是说，服务器总是针对每一行进行操作，而不是把它们看成一个整体。这使得处理大型数据集的效率很差。

下面这些针对触发器的通用警示条款也适用于 MySQL：

- 它们让服务器正在做的事情变得模糊，因为一个简单的语句也可能导致很多“不可见”的工作。例如，如果一个触发器更新了一个相关的表，那么受影响的行的数量就会加倍。
- 触发器很难调试，并且它也不利于分析性能。
- 触发器会引起不明显的死锁和锁等待。如果触发器失败了，原始查询也会失败，如果没意识到触发器的存在，就很难准确地解释错误码。

从性能上说，MySQL 触发器实现上最严重的问题就是按行操作数据。这有时让使用触发器维护汇总表和缓存表变得不可能，因为它们会变得非常慢。使用触发器代替周期性的大量更新语句的主要原因是它们能让数据随时都保持一致。

触发器也不能保持原子性。例如，一个更新 MyISAM 表的触发器发生了错误，但却无法回滚。假设在一个 MyISAM 表中使用 AFTER UPDATE 触发器更新另外一个 MyISAM 表，如果触发器发生错误，导致第二个表更新失败，但第一个表的更新却不会被回滚。

InnoDB 表上的触发器的所有操作都在一个事务中完成，所以触发器和引发它的操作是原子操作。然而，如果做约束验证的时候使用 InnoDB 上的触发器去检查另外一个表的数据，那么就要当心 MVCC 架构，因为如果不小心的话，就不会得到正确的结果。例如，假设使用 BEFORE INSERT 触发器校验另外一个表中是否有匹配的数据，但是在读取数据的时候如果没有在触发器中使用 SELECT FOR UPDATE，对表的并发更新就会导致不正确的结果。

我们并不是让你害怕触发器，相反地，它们非常有用，特别是对于约束、系统维护任务，以及在同步中保持非正则化数据。

也可以使用触发器记录每一行的改变。它对于定制的复制器的配置非常方便，可以用于断开连接、改变数据，然后再把改变合并在一起。一个简单的例子就是一群带着笔记本去上班的用户，他们的改变需要同步到主服务器上，然后主服务器需要把这些改变拷回各自的机器。这需要双向同步。触发器是构建这种系统的好方式。每台笔记本使用触发器把数据改变全部都记录到一个表中，然后使用定制的工具把这些改变同步到主服务器中。最后，普通的 MySQL 复制机制可以让笔记本和主服务器保持同步，服务器此时已经有了所有的改变。

有时甚至可以为 FOR EACH ROW 设计找到变通的办法。罗兰德·波曼(Roland Bouman)发现触发器里面 ROW_COUNT() 始终返回 1，只有 BEFORE 触发器的第一行除外。可以使用这种办法防止触发器代码对每一行都执行一遍，并且只对每个语句运行一次。这和每个语句的触发器不同。但是它对于模拟每个语句的 BEFORE 触发器是一种有用的技巧。这种行为其实是一个缺陷，并且会在某个时候被修复。所以应该小心地使用它，并且在升级服务器的时候应该验证它是否还能继续工作。下面的例子展示了如何使用这种方式：

```
CREATE TRIGGER fake_statement_trigger
BEFORE INSERT ON sometable
FOR EACH ROW
BEGIN
    DECLARE v_row_count INT DEFAULT ROW_COUNT();
```



```
IF v_row_count <> 1 THEN
    -- Your code here
END IF;
END;
```

5.2.3 事件

Events

MySQL 5.1 提供了一种新的存储代码：事件（Event）。它类似于定时任务（Cron Job），但内部机制却完全不同。你可以创建事件，它会在某个特定时间或时间间隔执行一次预先写好的 SQL 代码。通常的方式就是将复杂的 SQL 语句包装到一个存储过程中，然后调用一下即可。

事件和连接完全无关，它运行在一个独立的定时器线程上。事件不接受参数，也不返回值，也没有任何连接让它们可以得到输入或返回输出。如果激活了服务器日志，就可以在日志中看到它们执行的命令，但是很难分辨哪些命令是从事件中执行的。可以查看 INFORMATION_SCHEMA_EVENTS 表了解事件的状态，比如上次执行时间。

事件有和存储过程相同的顾虑，那就是让服务器做了多余的工作。事件自身的开销是很小的，但是它调用的 SQL 语句可能会对性能产生严重的影响。适合事件的任务包括周期性的维护工作、重新建立缓存和汇总表以模拟物化视图，或者保存用于监视和诊断的状态值。

下面是一个事件的例子，它每周都会针对特定的数据库运行一次某个存储过程（注 5）。

```
CREATE EVENT optimize_somedb ON SCHEDULE EVERY 1 WEEK
DO
CALL optimize_tables('somedb');
```

可以定义事件是否应该被复制到从服务器。在某些情况下，复制是合适的，但是对另外一些情况则未必。以前一个例子为例，比如想对所有从服务器运行 OPTIMIZE TABLE 操作，但是要知道如果所有从服务器同时执行这种操作的话，性能就会受到某些因素的影响（比如表锁）。

最后，如果一个周期性事件需要较长的时间才能完成，那么有可能在前一个操作还没完成的时候后一次操作又发生了。MySQL 不会为这种行为提供保护，所以需要自己定义互斥的代码。可以使用 GET_LOCK() 来保证每次只有一个事件在运行：

```
CREATE EVENT optimize_somedb ON SCHEDULE EVERY 1 WEEK
DO
BEGIN
    DECLARE CONTINUE HANDLER FOR SQLEXCEPTION
    BEGIN END;
    IF GET_LOCK('somedb', 0) THEN
        DO CALL optimize_tables('somedb');
    END IF;
    DO RELEASE_LOCK('somedb');
END
```

上面代码中的 CONTINUE 保证了即使存储过程抛出异常，锁也能得到释放。

尽管事件和连接无关，但它和线程有关。服务器有一个主调度线程，必须用配置文件或利用命令把它激活：

```
mysql> SET GLOBAL event_scheduler := 1;
```

注 5：稍后会介绍如何创建存储过程。

一旦激活了它，每执行一个事件，都会创建一个新线程。在事件代码内部，调用 `CONNECTION_ID()` 会返回一个唯一的值，这个值其实是线程 ID。可以通过查看服务器错误日志了解事件执行情况。

5.2.4 保存存储代码的注释

224

Preserving Comments in Stored Code

存储过程、存储函数、触发器和事件都含有大量的代码，有适当的注释会很有帮助。但是注释不会被保持在服务器内部，因为命令行的客户端把它们都去掉了（命令行客户端的这个特性很不好，但这就是现实）。

一个保存注释的有用技巧就是使用特定版本的注释方式，这时服务器会把它们看做潜在的可执行代码，比方说它们只能在某个版本或更高版本的服务器上执行。服务器和客户端程序知道它们不是普通的注释，所以就不会把它们丢掉。为了防止这些“代码”被执行，可以使用一个很高的版本数字，比如 99999。例如，下面就是一个给触发器加注释的例子：

```
CREATE TRIGGER fake_statement_trigger
BEFORE INSERT ON sometable
FOR EACH ROW
BEGIN
    DECLARE v_row_count INT DEFAULT ROW_COUNT( );
    /*!99999
       ROW_COUNT( ) is 1 except for the first row, so this executes
       only once per statement.
    */
    IF v_row_count <> 1 THEN
        -- Your code here
    END IF;
END;
```

5.3 游标

Cursors

MySQL 现在提供了只能向前的只读游标（Cursor），它只能在存储过程中使用。它可以迭代每一行结果，并且把结果提取到变量中以做进一步处理。一个存储过程在同一时刻可以使用多个游标，也可以在循环中嵌套游标。

MySQL 以后也许会提供用于更新的游标，但现在没有。因为游标读取的是临时表，而不是产生数据的原始表，所以它是只读的。

一不小心就会掉入游标的陷阱，它是在临时表上执行的，容易让开发人员产生对性能的误解。最重要的事情是要知道当打开游标的时候，它执行的是整个查询。看看下面的存储过程：

```
1 CREATE PROCEDURE bad_cursor( )
2 BEGIN
3     DECLARE film_id INT;
4     DECLARE f CURSOR FOR SELECT film_id FROM sakila.film;
5     OPEN f;
6     FETCH f INTO film_id;
7     CLOSE f;
8 END
```

这个例子说明了可以在对所有的结果进行迭代前关闭游标。习惯了 Oracle 和 Microsoft SQL Server 的用户不会

225

觉得这个过程有什么错误，但它引起了很多不必要的工作。使用 `SHOW STATUS` 命令分析这个存储过程，发现它进行了 1000 次索引读取和 1000 次插入。其原因是 `sakila.film` 表有 1000 行数据。所有这 1000 次操作都发生在代码的第 5 行。

这个例子的价值就是告诉你早早关闭一个从大型结果中读取数据的游标，不会节约任何的工作。如果只需要几行数据，就应该使用 `LIMIT`。

游标也会导致 MySQL 执行额外的 I/O 操作，这会非常慢。临时表位于内存中，它不支持 `BLOB` 和 `TEXT` 数据类型，MySQL 不得不在磁盘上为包含这些类型的游标创建临时表。如果没有这种情况，当临时表大于 `tmp_table_size` 时，MySQL 也会在磁盘上创建临时表。

MySQL 不支持客户端游标，但是客户端 API 通过把结果提取到内存中，可以模拟游标操作。这和在应用程序中把结果放入数组没什么区别。第 161 页的“MySQL 客户端/服务器协议”有更多把结果放到客户端内存对性能影响的内容。

5.4 准备语句

Prepared SQL Statement

MySQL 4.1 及其更高版本支持服务器端准备语句（Prepared Statements），它使用增强的二进制客户端/服务器协议在客户端和服务器之间高效地发送数据。可以通过支持这种新协议的编程库来访问准备语句，例如 MySQL C API。MySQL Connector/J 和 MySQL Connector/NET 为 Java 和 .NET 提供了同样的访问接口。它也有 SQL 语言的访问接口，稍后再讨论这一接口。

创建准备语句时，客户端库会向服务器发送一个实际查询的原型，然后服务器对该原型进行解析和处理，将部分优化过的原型保存起来，并且给客户端返回一个状态句柄（State Handle）。客户端可以通过定义状态句柄重复地执行查询。

准备语句可以有参数，它用问号（?）代表执行时的具体参数。比如，可以按下面的方式准备查询：

```
mysql> INSERT INTO tbl1(col1, col2, col3) VALUES (?, ?, ?) ;
```

接下来可以把状态句柄和每个问号对应的值发送到服务器执行查询。这个过程可以重复任意次。发送状态句柄的具体方式取决于编程语言。其中一种方式是利用 MySQL 为 Java 和 .NET 定制的连接器（Connector）。许多其他客户端类库也有类似的接口，实际使用时请查看具体文档。

使用准备语句会比多次执行查询效率高得多，具体原因如下：

- 服务器只需要解析一次查询，这节约了解析和其他的开销。
- 因为服务器缓存了一部分执行计划，所以它只需要执行某些优化步骤一次。
- 通过二进制发送参数比通过 ASCII 码要快得多。比如，通过二进制发送 `DATE` 类型的参数只需要 3 个字节，但通过 ASCII 码发送需要 10 个字节。节约的效果对于 `BLOB` 和 `TEXT` 类型最为显著，因为它们可以成块地发送，而不是一个个地发送。二进制协议也帮客户端节约了内存，同时减少了网络开销和数据从本身的类型转换为非二进制协议的开销。
- 整个查询不会被发送到服务器，只有参数才会被发送，这减少了网络流量。
- MySQL 直接把参数保存在服务器的缓冲区内，不需要在内存中到处拷贝数据。

准备语句对安全性也有好处。它不需要在应用程序中对值进行转义和加引号，这更加方便，并且减少了 SQL 遭受注入攻击和其他攻击的可能（永远也不能信任用户的输入，即使使用准备语句也不能）。

只有准备语句能使用二进制协议。使用普通的 `mysql_query()` 函数提交查询不会使用二进制协议。许多客户端类库能用问号“准备”查询，然后定义问号的值以执行查询，但是这些类库通常都是在客户端模拟准备语句的，而实际上是把查询用 `mysql_query()` 发送到服务器。

5.4.1 准备语句优化

Prepared Statement Optimization

MySQL 缓存了准备语句的部分执行计划，但是不会预先计算和缓存一些依赖于具体值的优化过程。按照优化执行的时机来划分，它们可以被分为 3 类。下面的列表在写作本书的时候是可用的，但是将来可能会有所不同。

准备时期

服务器解析查询文本、消除否定条件并且重写子查询。

第一次执行时

服务器简化嵌套联接并且把可以转化的外联接转化为内联接。

每次执行时

服务器此时会进行：

- 修整分区。
- 消除可以消除的 `COUNT()`、`MIN()` 和 `MAX()`。
- 移除常量子表达式。
- 探测常量表。
- 传递相等性。
- 分析和优化 `ref`、`range` 和 `index_merge` 等访问方法。
- 优化联接顺序。

要知道更多有关优化的内容，请查阅第 4 章。

5.4.2 准备语句的 SQL 接口

Prepared Statement SQL Interface

MySQL 4.1 及以上版本为准备语句提供了 SQL 语言接口。下面是一个具体的例子：

```
mysql> SET @sql := 'SELECT actor_id, first_name, last_name
-> FROM sakila.actor WHERE first_name = ?';
mysql> PREPARE stmt_fetch_actor FROM @sql;
mysql> SET @actor_name := 'Penelope';
mysql> EXECUTE stmt_fetch_actor USING @actor_name;
+-----+-----+-----+
| actor_id | first_name | last_name |
+-----+-----+-----+
|          1 | PENELOPE   | GUINNESS  |
|          54 | PENELOPE   | PINKETT   |
```

```

|          104| PENELOPE      | CRONYN      |
|          120| PENELOPE      | MONROE      |
+-----+-----+-----+
mysql> DEALLOCATE PREPARE stmt_fetch_actor;

```

228 服务器在接收到这些语句时，会对它们进行翻译，最终效果和通过客户端类库执行完全一样。这说明不需要使用特殊的二进制协议去创建和执行准备语句。

如上可见，准备语句的语法比起一般的 SELECT 语句要笨拙一些。那么按照这种方式使用准备语句的好处是什么呢？

它主要用于存储过程。在 MySQL 5.0 中，可以在存储过程中使用准备语句，并且语法和 SQL 接口类似。这意味着可以通过连接字符串在存储过程内部执行“动态 SQL”，增加灵活性。例如，下面是一个可以在特定的数据库中针对每一个表都调用 OPTIMIZE TABLE 函数的存储过程：

```

DROP PROCEDURE IF EXISTS optimize_tables;
DELIMITER //
CREATE PROCEDURE optimize_tables(db_name VARCHAR(64))
BEGIN
    DECLARE t VARCHAR(64);
    DECLARE done INT DEFAULT 0;
    DECLARE c CURSOR FOR
        SELECT table_name FROM INFORMATION_SCHEMA.TABLES
        WHERE TABLE_SCHEMA = db_name AND TABLE_TYPE = 'BASE TABLE';
    DECLARE CONTINUE HANDLER FOR SQLSTATE '02000' SET done = 1;
    OPEN c;
    tables_loop: LOOP
        FETCH c INTO t;
        IF done THEN
            CLOSE c;
            LEAVE tables_loop;
        END IF;
        SET @stmt_text := CONCAT("OPTIMIZE TABLE ", db_name, ".", t);
        PREPARE stmt FROM @stmt_text;
        EXECUTE stmt;
        DEALLOCATE PREPARE stmt;
    END LOOP;
    CLOSE c;
END//
DELIMITER ;

```

可以按照下面的方式调用该存储过程：

```
mysql> CALL optimize_tables('sakila');
```

另外一种在存储过程进行循环的方式如下：

```

REPEAT
    FETCH c INTO t;
    IF NOT done THEN
        SET @stmt_text := CONCAT("OPTIMIZE TABLE ", db_name, ".", t);
        PREPARE stmt FROM @stmt_text;
        EXECUTE stmt;
        DEALLOCATE PREPARE stmt;
    END IF;
UNTIL done END REPEAT;

```

229 这两种循环结构之间有重要的区别。REPEAT 为每次循环检查了两次循环条件。在这个例子中，因为它仅仅只检查了一个整型值，所以不会有太大的性能影响，但是对于更复杂的检查，代价可能会比较大。

对于准备语句使用 SQL 接口而言,通过连接字符串来引用表和数据库是一种较好的方式,这样就无须使用参数。其实也不能对数据库和表的名字进行参数化,因为它们本身就是固定的标识符。另外一种情况就是动态设定 LIMIT 子句,它也不能使用参数。

SQL 接口对于手工测试准备语句是有用的,但是在存储过程之外它就不那么有用了。因为这个接口通过 SQL,没有使用二进制协议,并且它不能真正地减少网络流量,当有参数的时候,就不得不启动另外的查询给变量赋值。在某些特殊的情况下,可以通过这个接口得到好处,例如要准备一个巨大的 SQL 语句,它没有参数,需要执行很多次。然而,如果认为准备语句使用 SQL 接口会节约工作的话,应该做实际的评测。

5.4.3 准备语句的局限

Limitations of Prepared Statements

准备语句有下面的一些局限:

- 准备语句只针对一个连接,所以另外的连接不能使用同样的句柄。出于同样的原因,一个先断开再重新连接的客户端会丢失句柄(连接池或持续连接会减轻这个问题)。
- 准备语句不能使用 MySQL 5.0 以前版本的缓存。
- 使用准备语句并不总是高效的。如果只使用一次准备语句,那么准备它花费的时间可能比执行一次平常的 SQL 语句更长。准备语句也需要在服务器和客户端之间进行额外的信息交互。
- 现在不能在存储函数内部使用准备语句,但是可以在存储过程中使用准备语句。
- 如果忘记销毁准备语句,那么就有可能引起资源泄漏。这会消耗相当多的服务器资源。同样,因为对存储语句的数量有一个全局性的限制,所以一个错误可能会干扰其他使用准备语句的连接。

5.5 用户自定义函数

User-Defined Functions

230

MySQL 支持用户自定义函数(UDF, User-Defined Functions)已经有很长时间了。UDF 和用 SQL 语句写的存储函数不同,它可以用任何支持 C 调用方式的编程语言来写。

用户自定义函数必须被编译,然后动态地和服务器进行链接,使之和平台相关,并且给予你极强的功能。UDF 运行速度很快,并且能访问很多操作系统提供的功能和类库。SQL 存储函数很适合简单操作,比如计算球面上两点之间的最远圆周距离。但是如果想发送网络包,就需要使用 UDF。同样,现在不能使用 SQL 语句构建聚集函数,但用 UDF 可以轻易地做到这一点。

UDF 的能力越强,使用它的责任就越重大。UDF 中的一个错误可以使服务器完全崩溃,破坏内存和数据。它就像那些有错误的 C 代码一样危害巨大。



提示:和 SQL 语言写成的存储函数不同,UDF 现在不能读写表,至少在调用上下文中不行。这意味着它对纯计算,或者是和外部的交互非常有帮助。MySQL 和外部资源的交互正在增加。布莱恩·阿克尔(Brian Aker)和帕特里克·加尔布雷思(Patrick Galbraith)创建的 UDF 是一个极好的例子,它可以用于和数据库缓存服务器(Memcached)进行沟通,网址:http://tangent.org/586/Memcached_Functions_for_MySQL.html。

如果使用了 UDF，升级 MySQL 的时候要仔细地检查版本间的不同，也许需要对它进行重新编译，甚至得做出改变才能和新版本兼容。同时要保证 UDF 是绝对线程安全的，它们在 MySQL 服务器进程里面运行，那是纯粹的多线程环境。

网上有很多已经预先编译好的 UDF 类库，也有很多示例说明了如何创建自己的 UDF。最大的 UDF 存储库在：<http://www.mysqludf.org>。

下面是用于计算复制速度的 NOW_USEC() 函数（请查看第 405 页的“复制有多快”）。

```
#include <my_global.h>
#include <my_sys.h>
#include <mysql.h>
#include <stdio.h>
#include <sys/time.h>
#include <time.h>
#include <unistd.h>

extern "C" {
    my_bool now_usec_init(UDF_INIT *initid, UDF_ARGS *args, char *message);
    char *now_usec(
        UDF_INIT *initid,
        UDF_ARGS *args,
        char *result,
        unsigned long *length,
        char *is_null,
        Char *error);
}

my_bool now_usec_init(UDF_INIT *initid, UDF_ARGS *args, char *message) {
    return 0;
}

char *now_usec(UDF_INIT *initid, UDF_ARGS *args, char *result,
               unsigned long *length, char *is_null, char *error) {

    struct timeval tv;
    struct tm* ptm;
    char time_string[20]; /* e.g. "2006-04-27 17:10:52" */
    char *usec_time_string = result;
    time_t t;

    /* Obtain the time of day, and convert it to a tm struct. */
    gettimeofday (&tv, NULL);
    t = (time_t)tv.tv_sec;
    ptm = localtime (&t);

    /* Format the date and time, down to a single second. */
    strftime (time_string, sizeof (time_string), "%Y-%m-%d %H:%M:%S", ptm);

    /* Print the formatted time, in seconds, followed by a decimal point
     * and the microseconds. */
    sprintf(usec_time_string, "%s.%06ld\n", time_string, tv.tv_usec);

    *length = 26;

    return(usec_time_string);
}
```

5.6 视图

Views

视图 (View) 是很普通的概念, MySQL 5.0 包含了这一特性。MySQL 视图就像一张表, 但其自身不会存储任何数据, 它的数据来自 SQL 查询语句。

本书不会介绍如何创建或使用视图, MySQL 手册包含了创建视图的细节, 而其他很多文档也解释了如何使用它。出于很多考虑, MySQL 把视图当成表来看待, 并且两者共享了同样的命名空间。但是 MySQL 处理它们的方式并不完全一样。例如, 不能在视图上创建触发器, 也不能用 DROP TABLE 命令删除视图。

重要的是理解视图的内部机制, 以及和查询优化器交互的方式, 否则就别指望通过它得到好的性能。下面通过 world 数据库展示一下如何使用视图:

```
mysql> CREATE VIEW Oceania AS
-> SELECT * FROM Country WHERE Continent = 'Oceania'
-> WITH CHECK OPTION;
```

执行视图最简单的方式就是执行其中的 SQL 语句, 然后把结果放到临时表里面。在查询中使用视图的时候, 就可以引用临时表里面的数据。为了解它如何工作, 请看下面的例子:

```
mysql> SELECT Code, Name FROM Oceania WHERE Name = 'Australia';
```

下面是服务器的执行过程, 临时表的名字只是出于演示的目的:

```
mysql> CREATE TEMPORARY TABLE TMP_Oceania_123 AS
-> SELECT * FROM Country WHERE Continent = 'Oceania';
mysql> SELECT Code, Name FROM TMP_Oceania_123 WHERE Name = 'Australia';
```

这种方式很明显有性能和优化方面的问题。一种较好的使用方式是重写查询, 将视图的 SQL 语句与查询语句合并。下面是合并后的例子:

```
mysql> SELECT Code, Name FROM Country
-> WHERE Continent = 'Oceania' AND Name = 'Australia';
```

MySQL 可以使用这两种执行方式, 它们分别调用了合并算法 (MERGE Algorithm) 和临时表算法 (TEMPTABLE (注 6) Algorithm)。MySQL 会优先考虑使用合并算法。它甚至可以合并嵌套的视图。可以通过 EXPLAIN EXTENDED 命令, 加上 SHOW WARNINGS 参数查看重写后的查询。

如果使用临时表算法, 解释器通常会把它显示为 DERIVED 表。图 5-4 显示了这两种执行方式。

在视图包含 GROUP BY、DISTINCT、聚集函数、UNION、子查询或其他无法保持视图返回的行和待查询的行之间一对一关系的结构的时候, MySQL 就会使用临时表算法。其实并非只有在上面的情况下才会使用临时表算法, 具体情况将来也许还会发生改变。如果想知道一个视图使用的是合并算法还是临时表算法, 使用解释器解释一下即可:

```
mysql> EXPLAIN SELECT * FROM <view_name>;
+-----+-----+
| id | select_type |
+-----+-----+
| 1 | PRIMARY |
| 2 | DERIVED |
+-----+-----+
```

注 6: 它指临时表 (Temp Table), 而不是指有诱惑力 (Can be tempted)。

选择类型是 DERIVED，这表明视图将会使用临时表算法。

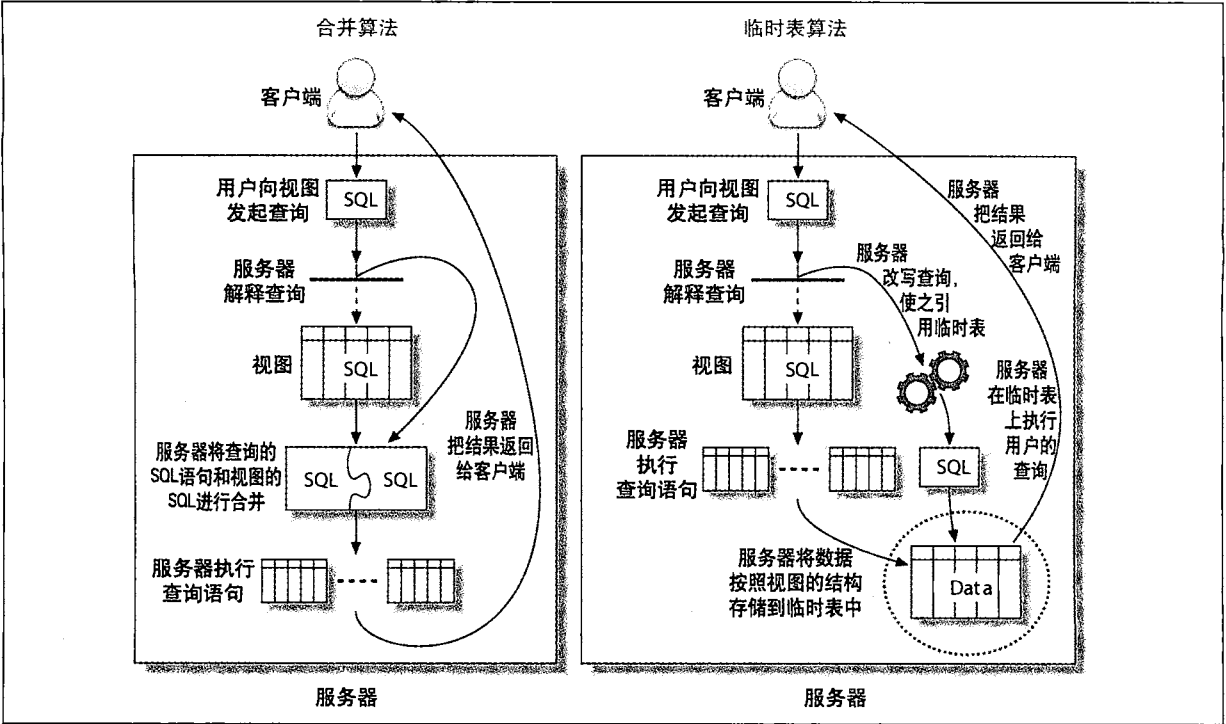


图 5-4: 视图的两种执行方式

5.6.1 可更新视图

Updatable Views

可更新视图可以通过视图更新它所引用的表。在某些特定的条件下，可以对视图使用 UPDATE、DELETE，甚至 INSERT 语句。下面的例子是一个有效的操作：

```
mysql> UPDATE Oceania SET Population = Population * 1.1 WHERE Name = 'Australia';
```

视图如果含有 GROUP BY、UNION、聚集函数或任何其他的异常，就不是可更新的。改变数据的查询也许包含了联接，但是被改变的列必须在同一个表里面。使用 TEMPTABLE 算法的视图是不可更新的。

上一节创建视图时使用的 CHECK OPTION 子句，保证了通过视图改变的行和视图中 WHERE 子句始终匹配。所以，不能改变 Continent 列，也不能插入有不同 Continent 的行，任何一种情况都会导致服务器报告下面的错误：

```
mysql> UPDATE Oceania SET Continent = 'Atlantis';
ERROR 1369 (HY000): CHECK OPTION failed 'world.Oceania'
```

一些数据库运行在视图上面使用 INSTEAD OF 触发器，因此可以精确地定义当语句修改视图数据时会发生什么。但是 MySQL 不支持在视图上面使用触发器。一些关于可更新视图的限制在将来会被解决掉，一种可能性是在表上面用不同的存储引擎构造合并表。这也许会成为使用视图既高效又有用的方式。

5.6.2 视图对性能的影响

—— 来自《MySQL 5.6 权威指南》第 394 页

许多人不认为视图可以改进性能，但是它确实可以提高性能，也可以用它来支持其他提高性能的方式。例如，利用视图重构数据库架构的某一阶段，可以在更改它访问的表的同时，使代码继续工作。

一些应用程序为每一个用户使用一个表，这通常是为了实现行级别安全性。一个和前面例子类似的视图能够在表内实现类似的安全性，并且打开表的次数会更少，这有助于提高性能。被很多用户使用的许多开源工程已经累积了上百万张表，它们可以从这种方式得益。下面有一个假想的博客系统数据库：

```
CREATE VIEW blog_posts_for_user_1234 AS
  SELECT * FROM blog_posts WHERE user_id = 1234
  WITH CHECK OPTION;
```

可以使用视图实现列级权限，但是却没有实际创建这些权限的开销。列级权限也会阻止查询被缓存。视图可以限制访问想要的列，但却不会导致这些问题：

```
CREATE VIEW public.employeeinfo AS
  SELECT firstname, lastname -- but not socialsecuritynumber
  FROM private.employeeinfo;
GRANT SELECT ON public.* TO public_user;
```

有时可以使用伪临时视图取得好的效果。这不用创建一个只在当前连接中存在的真正临时视图，而是用一种特殊的方式，比如在特定的数据库中，创建一个视图，在使用完之后就可以把它删除。接下来就可以在 FROM 子句中使用这个视图，就像在 FROM 子句中使用子查询一样。这两种方式在理论上是同样的，但是 MySQL 对视图有不同的代码库（Codebase），所以通过临时视图可能会得到更好的性能。下面是一个例子：

```
-- Assuming 1234 is the result of CONNECTION_ID( )
CREATE VIEW temp.cost_per_day_1234 AS
  SELECT DATE(ts) AS day, sum(cost) AS cost
  FROM logs.cost
  GROUP BY day;

SELECT c.day, c.cost, s.sales
FROM temp.cost_per_day_1234 AS c
  INNER JOIN sales.sales_per_day AS s USING(day);

DROP VIEW temp.cost_per_day_1234;
```

要注意到使用了连接 ID 作为唯一的后缀，以避免名字冲突。这种方式在应用程序崩溃并无法删除临时视图时可以比较方便地进行清理。请查看第 394 页的“丢失的临时表”了解更多细节。

使用了临时表算法的视图性能可能会很差（尽管它们比没有使用视图的同样查询性能会好一些）。MySQL 在优化外部查询的时候就使用递归的步骤执行了它，这时外部查询的优化还没有完成，所以它没有得到完全的优化。构造临时表的查询不会从外部查询中得到 WHERE 条件，并且临时表没有索引。下面仍然使用 temp.cost_per_day_1234 举一个例子：

```
mysql> SELECT c.day, c.cost, s.sales
-> FROM temp.cost_per_day_1234 AS c
->   INNER JOIN sales.sales_per_day AS s USING(day)
->   WHERE day BETWEEN '2007-01-01' AND '2007-01-31';
```

在服务器执行这个视图，把结果放到临时表，然后联接 sales_per_day 表的过程中，到底发生了什么呢？WHERE 子句中的 BETWEEN 限制没有被“推送”到视图中，所以视图会为表中的所有日期创建结果，这不会有什么问题，

235

因为服务器将会把临时表排在联接的第一位，所以联接就可以使用 `sales_per_day` 表的索引。但是，如果将这样的两个视图进行联接，那么就不会有任何索引的优化了。

如果想用视图来提高性能，那么总是要进行测试，至少也应该进行分析。甚至合并视图也会增加开销，并且很难预测视图如何影响性能。如果有性能问题，永远也不要猜测，一定要进行测量。

视图引入了一些并非只有 MySQL 才有的问题。视图会误导开发人员，认为它非常简单，但实际上它非常复杂。某些查询看上去是反复查询一个表，但实际是查询一个代价很高的视图，一个不理解它潜在复杂性的开发人员会对这个现象无动于衷。我们曾经看到过一个很简单的查询在解释器中输出了上百条记录，因为其中某些表其实是视图，而它们又引用了很多其他的表和视图。

5.6.3 视图的局限

Limitations of Views

MySQL 不支持物化视图 (Materialized View)。物化视图通常把结果存储在一个不可见的表里面，然后周期性地从原始数据对不可见表进行刷新。MySQL 也不支持索引视图 (Indexed View)，可以通过创建缓存表和汇总表模拟物化视图和索引视图。但是在 MySQL 5.1 中，可以使用事件来调度这些任务。

MySQL 对视图的实现有一些让人烦恼的地方。最大的问题是 MySQL 不会保持视图的原始 SQL 语句。如果试着使用 `SHOW CREATE VIEW` 命令把视图显示出来，然后对它进行编辑，你会非常惊讶。视图被展开成了规范化的内部格式，不能对它进行格式化、添加注释和缩进。

如果想按照创建视图时的格式来编辑视图，可以打开视图的 .frm 文件，最后一行就是视图的文本。如果有 FILE 权限，并且 .frm 文件是可读取的，那么就可以通过 `LOAD_FILE()` 函数加载文件的内容。加载完毕后，进行一些字符串的操作就可以把语句原封不动地提取出来了，这儿要感谢 Roland Bouman 的创新性工作：

```
mysql> SELECT
-> REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(
-> REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(REPLACE(
-> SUBSTRING_INDEX(LOAD_FILE('/var/lib/mysql/world/Oceania.frm'),
-> '\nsource=', -1),
-> '\\\\', '\\_'), '\\%', '\\%'), '\\\\\\', '\\\\'), '\\z', '\\Z'), '\\t', '\\t'),
-> '\\r', '\\r'), '\\n', '\\n'), '\\b', '\\b'), '\\\\', '\\\\'), '\\\\', '\\\\'),
-> '\\0', '\\0')
-> AS source;
+-----+
| source
+-----+
| SELECT * FROM Country WHERE continent = 'Oceania'
  WITH CHECK OPTION
|
+-----+
```

5.7 字符集和排序规则

Character Sets and Collations

字符集 (Character Sets) 是从二进制编码到规定符号集的映射。可以把它看作如何用位来代表特定的字符。排序规则 (Collations) 是字符集排序原则的集合。在 MySQL 4.1 及其后续版本中，每个基于字符的值都有字符集

和排序规则（注 7）。MySQL 对字符集和排序规则的支持是世界级的，但是它也会增加复杂性，在某些情况下还会有一些性能开销。

本节讨论了在大多数情况下所需的设置和功能。如果需要了解更多的细节，请查阅 MySQL 手册。

5.7.1 MySQL 如何使用字符集

How MySQL Uses Character Sets

字符集有几种排序规则，并且每种字符集都有默认的排序规则。某个字符集特定的排序规则不能被用于其他字符集。可以将字符集和排序规则合在一起使用，所以从现在开始，我们使用字符集代指它们。

MySQL 有不同的选项控制字符集。这些选项和字符集很容易让人迷惑，所以记住它们的区别：只有基于字符的值才有字符集。其余的任何东西都只是规定使用何种字符集来进行比较或其他操作。基于字符的值可以是存储在列中的值、查询中使用的字面常量、表达式的结果、用户变量等。

MySQL 的设置可以分为两类：创建对象时的默认设置，以及用于控制服务器和客户端沟通的设置。

创建对象时的默认设置

MySQL 对服务器的每个数据库、每个表都有默认的字符集和排序规则。这形成了创建列时影响其字符集的默认值的继承关系。反过来说，它也告诉了服务器列存储的值使用了何种字符集。

在继承的每一个层次，都可以显式地定义字符集或让服务器使用默认值：

- 当创建一个数据库的时候，它从服务器继承了 `character_set_server` 设置。
- 当创建表的时候，它从数据库继承字符集。
- 当创建列的时候，它从表继承字符集。

记住，MySQL 存储数据的唯一地方就是列，所以继承的较高层次只有默认值。表的默认字符集不会影响到表里面存储的数据，它只是告诉 MySQL 在创建列的时候如果没有指定字符集，那么就使用该默认值。

用于客户端/服务器沟通的设置

当服务器和客户端彼此进行沟通的时候，它们也许会使用不同的字符集来回传递数据。服务器将会按需进行翻译：

- 服务器假设客户端正在按照 `character_set_client` 设置的字符集发送数据。
- 服务器从客户端收到语句后，它按照 `character_set_connection` 设置的字符集对数据进行翻译，它也会用这个字符集决定如何把数字转换为字符串。
- 当服务器把结果或错误信息返回给客户端时，它会按照 `character_set_result` 定义的字符集进行翻译。

图 5-5 展示了这一过程。

注 7：MySQL 4.0 及之前的版本对整个服务器采用了全局设置，并且只能从几个 8 位字符集中进行选择。

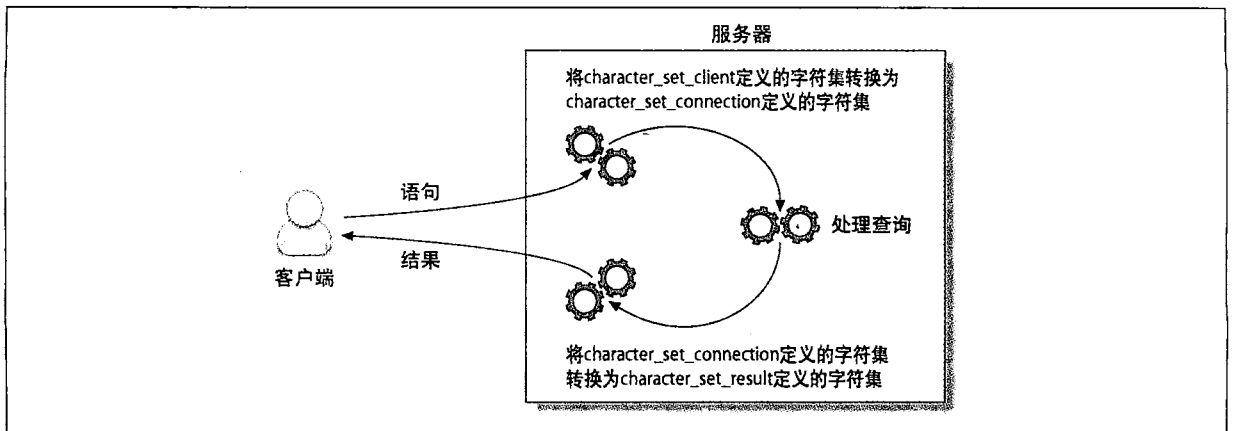


图 5-5：客户端和服务端字符集

可以使用 SET NAMES 命令，或者 SET CHARACTER SET 命令按照自己的需要改变上面的 3 个设置。但是，这两个命令只会影响服务器的设置。客户端程序和 API 也需要正确地进行设置，以避免和服务器的沟通问题。

239 假设使用 latin1 字符集打开了一个连接（这是默认的字符集，除非用 mysql_options() 改变了它），然后使用 SET NAMES utf8 告诉服务器客户端将会用 utf8 字符集发送数据。这样的话，就造成了字符集不匹配的问题，它会导致错误，甚至安全性问题。此时应该设置客户端字符集并在需要转义的时候使用 mysql_real_escape_string()。在 PHP 中，可以使用 mysql_set_charset() 改变客户端的字符集。

MySQL 如何比较值

当 MySQL 比较两个使用了不同字符集的值时，它必须把它们转换为同一字符集。如果字符集不兼容，就会引发错误，例如“错误 1267 (HY000)：不合法的排序规则混合。(ERROR 1267 (HY000): Illegal mix of collations)”在这种情况下，通常需要使用 CONVERT() 函数显式地把值转换为兼容的字符集格式。这个错误在 MySQL 4.1 中更为常见。

MySQL 也能对值采取强制措施。它决定了值采用的字符集的优先级并且影响了 MySQL 会隐式转换的值。可以使用 CHARSET()、COLLATION() 和 COERCIBILITY() 函数来调试和字符集及排序规则有关的问题。

可以使用导引符 (Introducer) 和排序规则子句 (Collate Clauses) 来定义字面常量的字符集和排序规则，如下例：

```
mysql> SELECT _utf8 'hello world' COLLATE utf8_bin;
+-----+
| _utf8 'hello world' COLLATE utf8_bin |
+-----+
| hello world                          |
+-----+
```

特殊行为

MySQL 字符集的行为有一些出人意料的地方。下面是一些值得注意的事项：

难以捉摸的 character_set_database 设置

character_set_database 的默认值是默认数据库的值。当改变默认数据库的时候，它也会跟着改变。如果没有默认数据库，它的默认值就是 character_set_server。

LOAD DATA INFILE

LOAD DATA INFILE 按照当前 character_set_database 的设置解释接收到的数据。一些版本的 MySQL 在 LOAD DATA INFILE 语句中接受可选的 CHARACTER SET 子句，但是不应该依赖于它。

取得可靠结果的最佳方式是用 USE 命令使用需要的数据库，并执行 SET NAMES 选择一个字符集，最后再加载数据。不管列的字符集是什么，MySQL 都会按照同一种字符集解释数据。

240

SELECT INTO OUTFILE

MySQL 利用 SELECT INTO OUTFILES 无转换地写入数据。如果不用 CONVERT 函数包装列，就无法设置数据的字符集。

嵌入的转义序列

MySQL 按照 character_set_client 的设置解释转义序列，即使有导引符或者排序规则子句也是这样。因为解析器是以字面常量的方式解析转义序列的。到目前为止，解析器没有排序的意识，它不会把导引符看成一种指令，而仅仅是一个标识。

5.7.2 选择字符集和排序规则

MySQL 4.1 及其以上版本支持大量的字符集和排序规则，包括使用 Unicode 字符集的 UTF-8 编码支持多字节字符（MySQL 支持完整的 UTF-8 三字节子集，它能存储世界上的绝大多数语言）。可以使用 SHOW CHARACTER SET 及 SHOW COLLATION 命令查看 MySQL 支持的字符集。

最常见的排序规则是利用大小写或字母的二进制编码进行排序。排序规则通常用 _cs、_ci 或_bin 结尾，这样就能很轻易地分辨它们。

在显式地定义字符集的时候，并不需要同时定义字符集和排序规则。如果忽略了其中一个，或者两个全部都被忽略了，MySQL 就会使用应用程序的默认值补上。表 5-2 显示了 MySQL 如何决定使用的字符集和排序规则。

表 5-2: MySQL 如何决定字符集和排序规则的默认值

| 如果定义 | 结果字符集 | 结果排序规则 |
|----------|-------------|------------|
| 字符集和排序规则 | 按照定义 | 按照定义 |
| 只有字符集 | 按照定义 | 字符集的默认排序规则 |
| 只有排序规则 | 拥有该排序规则的字符集 | 按照定义 |
| 都没有 | 可用的默认值 | 可用的默认值 |

下面的命令显示了如何利用特定的字符集和排序规则创建数据库、表和列。

```
CREATE DATABASE d CHARSET latin1;
CREATE TABLE d.t(
    col1 CHAR(1),
    col2 CHAR(1) CHARSET utf8,
    col3 CHAR(1) COLLATE latin1_bin
) DEFAULT CHARSET=cpl251;
```

241

结果表的列有下面的排序规则：

```
mysql> SHOW FULL COLUMNS FROM d.t;
+-----+-----+-----+
| Field | Type      | Collation      |
+-----+-----+-----+
| col1  | char(1)   | cp1251_general_ci |
| col2  | char(1)   | utf8_general_ci   |
| col3  | char(1)   | latin1_bin        |
+-----+-----+-----+
```

保持简单

在数据库中混合多种字符集确实是很糟糕的事情。不兼容的字符集会让人非常困惑。它们甚至只会在某些字符出现的时候才会出问题，到那时，所有的操作都会出错（比如表联接）。只能使用 ALTER TABLE 命令将列转换为兼容的字符集，或者在 SQL 语句中使用引导符或排序规则子句将值转化为需要的字符集。

为了让你的头脑保持清醒，最好的选择就是在服务器级选择合适的默认值，也许在数据库级也可以，然后就可以在列这一级具体问题具体分析。

5.7.3 字符集和排序规则如何影响查询

How Character Sets and Collations Affect Queries

一些字符集需要更多的 CPU 操作、消耗更多的内存和存储空间，甚至会使索引失效。因此要仔细地选择字符集和排序规则。

在字符集和排序规则之间做转化会增加某些操作的开销。例如，sakila.film 表在 title 列上有索引，它可以用来加速 ORDER BY 查询：

```
mysql> EXPLAIN SELECT title, release_year FROM sakila.film ORDER BY title\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: film
         type: index
possible_keys: NULL
         key: idx_title0
        key_len: 767
         ref: NULL
         rows: 953
       Extra:
```

但是，只有在索引的排序规则和查询定义的规则一致的情况下服务器才能使用索引。索引是按照列的排序规则进行排序的，在本例中它是 utf8_general_ci。如果想使用另外一种排序规则，服务器就不得不进行文件排序：

```
mysql> EXPLAIN SELECT title, release_year
-> FROM sakila.film ORDER BY title COLLATE utf8_bin\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: film
         type: ALL
possible_keys: NULL
         key: NULL
        key_len: NULL
         ref: NULL
         rows: 953
       Extra: Using filesort
```


除了适应联接的默认字符集和在查询中显式定义的字符集之外, MySQL 还需要转换字符集, 以进行比较。例如, 使用字符集不同的列联接两个表, MySQL 就不得不在它们之间做转换。这种转换使联接无法利用索引, 因为它就像使用一个函数对列进行封装一样。

UTF-8 多字节字符集以可变字节数 (1 字节到 3 字节) 来保存每个字符。MySQL 内部许多字符串操作使用了固定大小的缓冲区, 所以它必须分配足够的空间, 以容纳最大的可能长度。例如, CHAR(10) 使用 UTF-8 进行编码, 即使实际的字符串没有所谓的宽字符, 也需要 30 个字节。可变长度字段 (VARCHAR, TEXT) 保存在磁盘上时不会有这样的问题。但是内存中的临时表总是分配需要的最大长度。

在多字节字符集中, 一个字母不再是一个字节。因此, MySQL 有 LENGTH() 和 CHAR_LENGTH() 函数, 它们不会对多字节字符返回同样的结果。在使用多字节字符集的时候, 要确保在统计字符 (比如执行 SUBSTRING() 操作) 的时候使用 CHAR_LENGTH() 函数。这个问题同样也存在于应用程序语言中。

另外一个出人意料的地方就是索引限制。如果索引了一个使用了 UTF-8 字符集的列, MySQL 就会假设每个字符都会是 3 字节, 所以平常的长度限制突然就减少为三分之一:

```
mysql> CREATE TABLE big_string(str VARCHAR(500), KEY(str)) DEFAULT CHARSET=utf8;
Query OK, 0 rows affected, 1 warning (0.06 sec)
mysql> SHOW WARNINGS;
+-----+-----+-----+
| Level | Code | Message |
+-----+-----+-----+
| Warning | 1071 | Specified key was too long; max key length is 999 bytes |
+-----+-----+-----+
```

注意到 MySQL 自动地将索引的长度减少到 333 个字符。

```
mysql> SHOW CREATE TABLE big_string\G
***** 1. row *****
      Table: big_string
Create Table: CREATE TABLE `big_string` (
  str` varchar(500) default NULL,
  KEY `str` (`str`(333))
) ENGINE=MyISAM DEFAULT CHARSET=utf8
```

如果没有注意到警告信息并且检查表的定义, 就不会认识到索引仅仅是列的前缀。这会有一些副作用, 例如会禁止覆盖索引。

某些人推荐在所有的地方都使用 UTF-8。但是, 如果在意性能的话, 这不是一个好主意。许多应用程序根本不需要 UTF-8, 并且使用何种字符集依赖于数据。UTF-8 使用更多的磁盘空间。

在决定字符集的时候, 重要的是考虑将要存储的数据。例如, 如果存储英文文本, UTF-8 不会有问题, 因为大部分英文字符在 UTF-8 里面都是用一个字节存储的。在另一方面, 如果保存阿拉伯文或俄文这样的非拉丁语言, 情况就会很不一样。一个只需要存储阿拉伯语的应用程序语言可以使用 cp1256 字符集, 它可以用一个字节包含所有的阿拉伯字符。但是如果程序需要保存许多不同的语言并且选择了 UTF-8, 那么同样的阿拉伯字符就会使用更多的存储空间。同样地, 如果将某个国家的字符集转换为 UTF-8, 存储空间将会极大地上升。如果使用 InnoDB, 就有可能把数据大小增加到超出页面限制的地步, 并且需要外部存储空间。这会造成存储空间大量浪费并且导致碎片。更多细节请参阅第 298 页的“优化 BLOB 和 TEXT 工作负载”。

有时根本不需要使用字符集。字符集对于不区分大小写比较、排序, 以及 SUBSTRING() 这样的字符串操作非常有用。如果不需要数据库服务器处理字符, 就可以把任何数据, 包括 UTF-8 数据, 存储在 BINARY 列中。如果

这么做的话，就需要加一列标明用来编码的字符集。尽管有些人使用这种方法已经很久了，但是还是需要更小心一点。如果你忘记了一个字节并不一定是一个字符，它就能导致很难捕获的错误，比如和 `SUBSTRING()` 和 `LENGTH` 相关的错误。在实际工作中，我们建议你最好避免这种方式。

5.8 全文搜索

大部分的查询都可能包含 `WHERE` 子句，用于比较相等性、过滤数据等。但是，也许还需要执行关键字搜索，它基于数据的关联性，而不是相互比较。全文搜索系统就是为这个目的而设计的。

全文搜索 (Full-Text Searching) 需要特殊的查询语法。无论有没有索引，它都能工作，但是索引可以加速匹配的过程。全文搜索使用的索引有特殊的结构，可以帮助发现含有所要关键字的文档。

你也许不知道全文索引，但是你至少知道一种全文索引引擎：因特网搜索引擎。尽管它们规模很大，而且后端并不使用关系数据库，但是基本原则是一样的。

在 MySQL 中，只有 MyISAM 存储引擎支持全文索引。可以在上面搜索基于字符的内容 (`CHAR`、`VARCHAR` 和 `TEXT` 列)，并且它支持自然语言搜索和布尔搜索。全文搜索的实现有很多限制和缺陷 (注 8)，并且它非常复杂，但是它仍然被广泛地使用，因为它被包含在服务器中，而且能满足许多应用程序的要求。本节广泛地讨论了如何使用它，以及如何在有全文搜索的情况下，为得到好的性能而进行设计。

MyISAM 全文索引操作了一个全文集合 (Full-text Collection)，它由单个表中的一个或多个字符列组成。实际上，MySQL 在集合中通过联接列构造索引，并且把它们当成很长的字符串进行索引。

MyISAM 全文索引是一种特殊的具有两层结构的 B 树。第 1 层保存了关键字，然后，对每个关键字，第 2 层包含了一个列表，它由相关的文档指针组成，这些指针指向包含该关键字的全文集合。索引不会包含集合中的每一个词，它按照下面的方式进行调整：

- 一个停用字 (Stopword) 清单把某些无意义的词剔除了，这样它们就不会被索引。停用字列表基于常用的英语用法，但是可以使用 `ft_stopword_file` 选项用一个外部列表替换掉它。
- 除非一个词长度大于 `ft_min_word_len` 并且小于 `ft_max_word_len`，否则它就会被忽略掉。

全文索引没有存储关键字发生的列信息，所以如果要对组合的列进行搜索，就要创建多个索引。

这也意味着不能使用 `MATCH AGAINST` 子句来指定某些列里面的词比另外列里面的词更重要。这在构建网站的搜索引擎时其实是很常见的。比如，我们可能会需要当关键字出现在标题里面时，结果就显示在前面。如果需要这种特性，就不得不写一些更复杂的查询 (稍后会有示例)。

5.8.1 自然语言全文搜索

自然语言搜索查询决定了每个文档和查询的关联性。关联性基于匹配的单词数量，以及它们在文档中出现的频率。

注 8：也许会发现 MySQL 的全文搜索引擎的限制让它在应用程序中根本就不可用。在附录 C 中讨论了使用 Sphinx 作为外部的全文搜索引擎。

在所有索引中出现得越少的单词，说明搜索的关联性越高。相反，极其常见的单词根本就不值得搜索。对于那些在多于一半的行里面出现的单词，即使它们没有被列在停用字列表（Stop Words List）里面，自然语言全文搜索也会把它们排除掉（注9）。

全文搜索的语法和其他类型的查询有一点点不同。可以通过使用带有 MATCH AGAINST 的 WHERE 子句告诉 MySQL 执行全文搜索。下面是一个例子。在标准的 sakila 数据库里面，film_text 表在 title 和 description 列上面有全文索引：

```
mysql> SHOW INDEX FROM sakila.film_text;
+-----+-----+-----+-----+
| Table      | Key_name                | Column_name | Index_type |
+-----+-----+-----+-----+
| ...
| film_text  | idx_title_description    | title       | FULLTEXT   |
| film_text  | idx_title_description    | description  | FULLTEXT   |
+-----+-----+-----+-----+

mysql> SELECT film_id, title, RIGHT(description, 25),
-> MATCH(title, description) AGAINST('factory casualties') AS relevance
-> FROM sakila.film_text
-> WHERE MATCH(title, description) AGAINST('factory casualties');
+-----+-----+-----+-----+
| film_id | title                    | RIGHT(description, 25) | relevance |
+-----+-----+-----+-----+
| 831     | SPIRITED CASUALTIES      | a Car in A Baloon Factory | 8.4692449569702 |
| 126     | CASUALTIES ENCINO        | Face a Boy in A Monastery | 5.2615661621094 |
| 193     | CROSSROADS CASUALTIES    | a Composer in The Outback | 5.2072987556458 |
| 369     | GOODFELLAS SALUTE        | d Cow in A Baloon Factory | 3.1522686481476 |
| 451     | IGBY MAKER               | a Dog in A Baloon Factory | 3.1522686481476 |
+-----+-----+-----+-----+
```

MySQL 会将搜索字符串拆成单词，然后逐个地和 title 及 description 字段进行匹配，这两个字段已经按照自己的索引被加入到了全文集合中。要注意到只有一个结果包含了这两个单词，并且 3 个包含“casualties”的结果被首先列了出来（整个表中也只有这 3 个含 casualties 的结果）。这是因为索引是按照相关性从高到低排列的。

246



提示：和普通的查询不同，全文搜索结果自动地按相关性进行了排序。MySQL 不能在进行全文搜索的时候使用索引来排序。因此，如果想避免文件排序，就不要使用 ORDER BY 子句。

就像上例中显示的那样，MATCH() 函数实际返回表示相关性的浮点数。可以用这个值按照相关性进行过滤或者在用户界面中显示。如果定义了 MATCH() 函数两次，也不会有额外的开销，MySQL 知道它们是同样的函数，只会执行一次。但是，如果在 ORDER BY 子句中使用了 MATCH()，MySQL 会使用文件排序来对结果进行排序。

必须在 MATCH 子句中明确地表示列，不管这些列是否在全文索引中被定义，或者 MySQL 根本就没使用索引。这是因为索引没有记录关键字在哪个列中出现过。

正如前文，这也意味着不能使用全文搜索定义某个关键字在特定的列中出现。但是也有变通的办法。可以在某些列上添加多个索引，然后利用索引的组合进行自定义排序。假设想让 title 列更重要，那么可以在这个列上再

注9：在测试中常见的错误就是只在全文搜索索引中加入很少的数据行，这样查询就不会找到任何结果。这是因为每个单词都在超过了一半的行中出现过。

加一个索引：

```
mysql> ALTER TABLE film_text ADD FULLTEXT KEY(title) ;
```

现在可以两次使用 title 来进行想要的排序：

```
mysql> SELECT film_id, title, RIGHT(description, 25),
-> ROUND(MATCH(title, description) AGAINST('factory casualties'), 3)
-> AS full_rel,
-> ROUND(MATCH(title) AGAINST('factory casualties'), 3) AS title_rel
-> FROM sakila.film_text
-> WHERE MATCH(title, description) AGAINST('factory casualties')
-> ORDER BY (2 * MATCH(title) AGAINST('factory casualties'))
-> + MATCH(title, description) AGAINST('factory casualties') DESC;
+-----+-----+-----+-----+
| title | RIGHT(description, 25) | full_rel | title_rel |
+-----+-----+-----+-----+
| 831 | a Car in A Baloon Factory | 8.469 | 5.676 |
| 126 | Face a Boy in A Monastery | 5.262 | 5.676 |
| 299 | jack in The Sahara Desert | 3.056 | 6.751 |
| 193 | a Composer in The Outback | 5.207 | 5.676 |
| 369 | d Cow in A Baloon Factory | 3.152 | 0.000 |
| 451 | a Dog in A Baloon Factory | 3.152 | 0.000 |
| 595 | a Cat in A Baloon Factory | 3.152 | 0.000 |
| 649 | nizer in A Baloon Factory | 3.152 | 0.000 |
```

但是这通常效率不高，因为它使用了文件排序。

5.8.2 布尔全文搜索

在布尔搜索中，查询自身定义了匹配单词的相对相关性。布尔搜索使用了停用词表（Stopword List）来过滤无用的单词，但是要禁用单词的长度必须大于 ft_min_word_len 且小于 ft_max_word_len 这一选项。布尔搜索的结果是没有排序的。

在构造一个布尔搜索查询的时候，可以使用前缀来修改搜索字符串中每个关键词的相对排名。最常用的修饰符在表 5-3 中。

表 5-3：布尔全文搜索常用修饰符

| 示例 | 含义 |
|-----------|----------------------|
| dinosaur | 含有“dinosaur”的行排名较高 |
| ~dinosaur | 含有“dinosaur”的行排名较低 |
| +dinosaur | 行必须含有“dinosaur” |
| -dinosaur | 行不能含有“dinosaur” |
| dino* | 含有以“dino”打头的单词的行排名较高 |

也可以使用其他的操作符，比如使用括号进行分组。可以用这种方式构造复杂的搜索。

还是举一个例子，仍然搜索 sakila.film_text 表，找到含有“factory”和“casualties”的电影。自然语言搜索会返回含有其中一个单词或包含了这两个单词的结果。但是，这儿使用的布尔搜索要求结果同时包含这两个单词：

```
mysql> SELECT film_id, title, RIGHT(description, 25)
-> FROM sakila.film_text
-> WHERE MATCH(title, description)
-> AGAINST('+factory +casualties' IN BOOLEAN MODE);
```

| film_id | title | right(description, 25) |
|---------|---------------------|---------------------------|
| 831 | SPIRITED CASUALTIES | a Car in A Baloon Factory |

也可以把单词用引号引起来，执行短语搜索，这要求精确匹配该短语。

```
mysql> SELECT film_id, title, RIGHT(description, 25)
-> FROM sakila.film_text
-> WHERE MATCH(title, description)
-> AGAINST('"spirited casualties"' IN BOOLEAN MODE);
```

| film_id | title | right(description, 25) |
|---------|---------------------|---------------------------|
| 831 | SPIRITED CASUALTIES | a Car in A Baloon Factory |

短语搜索很慢。只靠全文索引无法响应这种搜索，因为索引没有在原始的全文集合中记录单词之间的相对位置。这样造成的结果就是服务器不得不到行内部去执行单词搜索。

为了执行这种搜索，服务器将会查找所有含有“spirited”和“casualties”的文档。然后它会从这些文档中提取行，并且精确地匹配该短语。因为它使用了索引查找最开始使用的文档，所以你可能认为这会很快，至少比 LIKE 操作快得多。实际上，只要该短语并不常见，而且不会返回很多结果，它确实很快。如果短语非常常见，LIKE 实际会快一些，因为它会顺序读取数据，而不会使用索引排序的二次算法，并且它根本就不需要读取全文索引。

布尔全文搜索实际不需要全文索引。如果有全文索引的话，它就会使用索引，如果没有的话，它就会扫描整个表。甚至可以对多个表使用布尔全文搜索，例如对联接的结果进行搜索。但是在所有的情况下，它都很慢。

5.8.3 MySQL5.1 及以上版本中全文搜索的变化

MySQL5.1 引入了好些关于全文搜索的新变化。其中包括性能改进和构建用于改进内建功能的可插拔解析器。例如，插件可以改变索引的工作方式。它们可以比默认设置更灵活地把文本分割成单词，例如可以把 C++ 当成一个单词，还可以做预处理，索引不同类型的内容（比如 PDF），或者进行单词填充。插件还可以影响搜索工作的方式，比如通过填充搜索词条的方式。

InnoDB 的开发人员正在改进全文索引，但是不知道什么时候才能完成。

5.8.4 全文搜索的折中和变通方式

MySQL 对全文搜索的实现有一些设计上的局限。这些局限可能和特定的目的相冲突，但是也有很多办法可以绕过这些局限。

例如, MySQL 全文索引只会按照频率进行相关性排序。索引不会记录被索引的单词在字符串中的位置, 所以即使单词相邻, 也不会对相关性有贡献。尽管这对大多数应用都没有问题, 尤其是数据量较小的时候, 但是这仍然有可能无法达到需求, 并且 MySQL 全文索引没有提供自由选择排名算法的可能。它甚至没有把用于相邻性排名的数据保存起来。

数据大小是另外一个问题。MySQL 全文索引在数据大小和内存相匹配的时候工作得很好, 但是如果索引没有在内存中, 性能就会非常差, 对于很大的字段尤其如此。在使用短语搜索的时候, 数据和索引都必须和内存相匹配, 才能得到良好的性能。和其他索引类型比起来, 对全文索引进行添加、更新和删除都很慢。

- 修改有 100 个单词的文本需要的不是 1 次索引操作, 而是 100 次。
- 字段的长度通常对其他索引类型没什么影响, 但是对于全文索引, 有 3 个单词的文本和 1 万个单词的文本性能的差异可以相差几个数量级。
- 全文搜索索引更容易引起碎片, 并且要求更频繁地使用 `OPTIMIZE TABLE`。

全文索引也会影响服务器优化查询的方式。索引选择、WHERE 子句和 ORDER BY 工作的方式和我们期望的不一样:

- 如果有全文索引并且查询的 `MATCH AGAINST` 子句可以使用它, MySQL 会使用全文索引来处理查询。它并不会在索引之间做比较, 也许有其他的索引比全文索引更好, 但是 MySQL 不会考虑它们。
- 全文搜索索引只能做全文匹配。查询中的任何其他准则, 比如 WHERE 子句, 都必须在 MySQL 读取了行之后才能使用。而其他类型的索引可以一次性地检查 WHERE 子句的几个部分, 这也是一种区别。
- 全文索引不会存储索引的实际文本。因此, 不能像使用覆盖索引那样使用它。
- 全文索引不能被用于排序, 只能在使用自然语言搜索的时候按照相关性进行排序。如果想使用相关性以外的排序方式, MySQL 会使用文件排序。

250 让我们看看这些限制是如何影响查询的。假设有 100 万个文档, 文档的作者有普通索引, 内容有全文索引。现在想对文档内容做全文搜索, 但是只限于作者 123 的作品, 那么可能会写出下面的查询:

```
... WHERE MATCH(content) AGAINST ('High Performance MySQL')
      AND author = 123;
```

但是这个查询的性能会很差。MySQL 将会首先搜索开始的 100 万个文档, 因为它会首先应用全文索引。然后它再使用 WHERE 子句过滤掉其他作者的作品, 但是过滤的时候不能使用对作者的索引。

一个变通的办法就是在全文索引中包含作者的 ID。可以选择一个很不可能出现在文本中的前缀, 然后把作者的 ID 拼接到它后面, 并且把这个单词放到一个用于过滤的列中, 单独进行维护 (比如使用触发器)。

然后可以让全文索引包含这个过滤列, 并且重写查询如下:

```
... WHERE MATCH(content, filters)
      AGAINST ('High Performance MySQL +author_id_123' IN BOOLEAN MODE);
```

如果作者的 ID 非常具有选择性的话, 这种方式效率很高, 因为 MySQL 能很快地在全文索引中查找 “author_id_123”, 并且缩小查找范围。但是如果 ID 没有选择性, 性能可能会变得更差。要小心地使用这种方法。

有时可以使用全文索引来进行有边界的查询。比如, 想搜索固定范围内的座标, 就可以把座标编码到全文集合中。假定某行的座标是 X=123, Y=456。可以交错地把最重要的数字放在前面, 比如 XY142536, 然后把它放到一个列里面并用全文索引包含这个列。现在如果想进行一次限定了 X 和 Y 的矩形搜索, 比如 X 在 100 到 199

之间, Y 在 400 到 499 之间, 那么这时候就可以在查询中加上 “+XY14”。这比使用 WHERE 子句过滤要快得多。

有种技巧有时能很好地利用全文索引, 特别是对于分页显示, 就是通过全文索引选择主键的列表并且缓存结果。当应用程序准备好了渲染结果时, 它可以用另外一个查询通过主键提取想要的列。第二个查询可以包括更多负责的判断条件或联接, 它们可以更好地使用其他的索引。

尽管只有 MyISAM 支持全文索引, 如果想使用 InnoDB 或其他存储引擎, 也不用担心, 因为你可以自己做全文索引。一个通常的办法就是把表复制到从服务器上, 它可以使用 MyISAM 存储引擎, 然后使用从服务器进行全文搜索。如果不想在不同的服务器上处理查询, 可以把表垂直地分为两部分, 一部分保留文本列, 另一部分保留其余的数据。

也可以把某些列复制到被全文索引了的表中。Sakila.film 表使用了这种技巧, 那些列是用触发器维护的。但是另外一种替代的办法就是使用外部的全文引擎, 例如 Lucene 和 Sphinx。附录 C 有 Sphinx 的更多内容。

使用了全文搜索的 GROUP BY 查询简直就是性能杀手。这是因为全文搜索会找到大量的匹配数据, 引起了随机磁盘读取, 然后使用临时表或文件排序来进行分组。因为这些查询通常都是为了寻找每个分组中靠前的数据, 一种比较好的优化方式就是对数据进行抽样, 而不是完全精确地匹配。例如, 选择最开始的 1 000 行, 放入临时表中, 然后为每个分组返回靠前的数据。

5.8.5 全文调优和优化

Full-Text Tuning and Optimization

对全文索引进行常规的维护是提高性能的最重要方式。全文索引使用的是双重平衡树结构 (Double B-Tree), 并且还有大量的关键词, 这意味着它比普通的索引更容易出现碎片。常常要使用 OPTIMIZE TABLE 来去除索引的碎片。如果服务器是 IO 密集型的, 那么这会比周期性删除和重建索引要快得多。

对于需要键缓冲区 (Key Buffers) 的全文搜索来说, 如果缓存能容纳全文索引, 那么服务器的性能就会更好, 因为全文索引都在内存中的话, 工作效率就会更高。实际上可以使用专用的键缓冲区, 以免别的索引把全文索引冲掉。请参阅第 274 页的 “MyISAM 键缓存” 以了解更多细节。

提供好的停用词表也非常重要。默认的对普通的英语文本效果不错, 但是它未必适用于别的语言或专门的技术文档。例如, 在索引有关 MySQL 的文档时, mysql 应该就是停止字, 因为它出现的实在太频繁了, 对搜索没什么帮助。

跳过短的单词常常能提高性能。长度可以使用 ft_min_word_len 进行配置。增加长度会跳过更多的单词, 使索引变得更小和更快, 但是精确性也会降低。在某些特别的情况下, 也许会需要很短的单词。例如, 一个在消费电器文档中搜索 “cd player” 的查询可能会产生大量无关的结果, 除非索引允许短的单词。搜索 “cd player” 的用户不想在结果中看到 MP3 和 DVD 的信息, 但是最短长度默认是 4, 所以搜索只会查找 “player”, 这样会返回大量的无关的播放器信息。

停止词表和最短长度可以把某些单词排除在搜索之外, 但是搜索的质量也会受到它们的影响。正确的平衡依赖于具体的应用。如果需要好的性能和好的搜索结果, 那么就要按照程序的要求自己定制这两个参数。一个较好的方式就是用日志进行记录, 然后调查常见的搜索、不常见的搜索、不会返回结果的搜索以及会返回许多结果的搜索。可以用这种方式了解用户需求和搜索的内容, 然后就可以利用这些信息来改进搜索的性能和质量。



提示：要知道如果改变了单词的最小长度，那么可能就需要使用 `OPTIMIZE TABLE` 重建索引，以使其生效。一个相关的参数是 `ft_max_word_len`，它通常可以用来防止索引很长的关键词。

如果正在向数据库导入大量的数据并且希望全文索引某些列，那么在导入之前就应该使用 `DISABLE KEYS` 禁止全文索引，然后在导入后再用 `ENABLE KEYS` 重新启用它。这通常会更快，因为为插入的每一行更新索引需要很多时间，并且这样还可以避免碎片产生。

对于大型的数据集，可能需要手工地对它们进行分区，然后并行地进行搜索。这是很困难的任务，这时使用外部的全文搜索引擎可能会更好，例如 `Lucene` 和 `Sphinx`。经验表明它们的效率要高几个数量级。

5.9 外键约束

Foreign Key Constraints

InnoDB 现在是主要的支持外键的存储引擎。支持外键的选择目前看来比较有限（注 10）。MySQL 承诺过服务器自身会在某天提供和存储引擎无关的外键支持，但是目前看来，主要还是依靠 InnoDB 支持外键。所以我们这儿的讨论集中在 InnoDB 上。

外键是有开销的。在更改数据的时候，它们通常都会要求服务器查看另外一个表。尽管 InnoDB 有索引可以让这个操作更快，但是这通常不会消除这种检查的影响。它甚至能导致很大的索引，但是却毫无选择性。例如，假设一个巨大的表中有一个 `status` 列，并且想约束 `status` 的值必须是有效的，但是它只有 3 个可能的值。在这种情况下，尽管列本身很小，外键要求的额外索引也能极大地增加表的总大小，尤其是如果主键很大的话，更容易发生这种情况。这些索引除了进行外键检查之外没有任何用处。

但是，外键在某些情况下能改进性能。如果必须保证两个表有连续的数据，外键能让服务器进行更有效的检查。外键对于级联删除及更新也很有用。然而它们是逐行操作的，所以比起多表删除或批处理要慢一些。

外键使查询可以“到达”其他的表，这意味着需要锁。例如，向一个子表插入行，外键约束就会要求 InnoDB 检查父表中相关的内容。这肯定会把父表中的行锁住，以保证在事务完成前数据不会被删除掉。这导致了出乎意料的锁等待，甚至会引起死锁。这种问题很难进行调试。

有时可以用触发器来代替外键。外键对于级联更新这样的任务比触发器更好，但是外键只用于一种约束，就像前面的 `status` 列的例子，可以用触发器进行重写，并且显式地列出允许的值（可以使用 `ENUM` 数据类型），以达到较好的效率。

如果不把外键看成一种约束，那么用它来限制应用程序里的值通常是个好主意。

5.10 合并表和分区

Table Partitioning and Merging

合并表和分区是相关的概念，并且它们之间的区别会让人很迷惑。合并表是 MySQL 的一种特性，它可以把多个 MyISAM 表合并成一个虚表，就像对表使用了 `UNION` 的视图一样。可以使用合并存储引擎创建合并表。合并

注 10：PBXT 也支持外键。

表其实并不是一个真正的表，它更像一个用于放置相似表的容器。

相反的是，分区表是一个正常的表，它包含了一些特殊的指令，告诉 MySQL 物理数据被存放在什么地方。一个秘密就是分区表使用的存储字和合并表使用的极其相似。事实上，每个分区实际都是有索引的独立表，分区表其实包装了很多句柄对象。分区表看上去像一个单独的表，但它实际上是一大堆独立的表，但是无法访问分区表下面的独立表，不过合并表可以。

分区是 MySQL 5.1 的一种新特性。但是合并表已经有很长的历史了。两个特性都会带来同样的好处，它们可以让你做到下面这些事情：

- 分离静态的和变化的数据。
- 使用相关数据的物理相邻性来优化查询。
- 设计表以便查询访问较少的数据。
- 更容易地维护非常多的数据。（合并表在这个领域上比分区表更有优势）

因为 MySQL 在对分区表和合并表的实现上有很多共通之处，它们也有同样的限制。例如，分区表或合并表都有实际的限制，限制它们可以使用多少子表。在大部分情况下，几百张表就能造成性能下降。当我们具体分析某个系统的时候，会说明它的具体限制。

5.10.1 合并表

如果愿意的话，可以把合并表看成一种较老的、有更多限制的分区表，但是它们也有自己的用处，并且能提供一些分区表不能提供的功能。

合并表实际是容纳真正的表的容器。可以使用特殊的 UNION 语法来 CREATE TABLE。下面是一个合并表的例子：

```
mysql> CREATE TABLE t1(a INT NOT NULL PRIMARY KEY)ENGINE=MyISAM;
mysql> CREATE TABLE t2(a INT NOT NULL PRIMARY KEY)ENGINE=MyISAM;
mysql> INSERT INTO t1(a) VALUES(1),(2);
mysql> INSERT INTO t2(a) VALUES(1),(2);
mysql> CREATE TABLE mrg(a INT NOT NULL PRIMARY KEY)
-> ENGINE=MERGE UNION=(t1, t2) INSERT_METHOD=LAST;
mysql> SELECT a FROM mrg;
+-----+
| a     |
+-----+
| 1     |
| 1     |
| 2     |
| 2     |
+-----+
```

注意到合并表包含的表列的数量和类型都是一样的，并且合并表上的索引也会在下属表上存在。这是创建合并表的要求。也要注意在每个表的独有列上有主键，这会导致合并表有重复的行。这是合并表的一个局限：合并表内的每个表行为都很正常，但是它不会对下面的所有表进行强制约束。

INSERT_METHOD=LAST 指令告诉 MySQL 把所有的 INSERT 语句都发送到合并表的最后一个表上。定义 FIRST 或 LAST 是控制插入数据位置的唯一方式（但是也可以直接插入到下属表中）。

分区表可以更多地控制数据存放的位置。

下面的 INSERT 语句对合并表和下属表都可见：

```
mysql> INSERT INTO mrg(a) VALUES(3);
mysql> SELECT a FROM t2;
+----+
| a |
+----+
| 1 |
| 2 |
| 3 |
+----+
```

合并表还有其他有趣的特性和限制，比如删除合并表或它的某个下属表。删除合并表让所有的“子表”都变得不可访问，但是删除其中的某个子表有不同的影响，它的行为和操作系统有关。例如，在 GNU/Linux 上，子表的文件描述符还保持开启的状态，并且表还继续存在，但是只能从合并表中访问。

```
mysql> DROP TABLE t1, t2;
mysql> SELECT a FROM mrg;
+-----+
| a      |
+-----+
|      1 |
|      1 |
|      2 |
|      2 |
|      3 |
+-----+
```

还有一些另外的局限性和特殊行为。最好的办法是阅读手册，但是在这儿要说的是 REPLACE 并不能在所有的合并表上工作，并且 AUTO_INCREMENT 不会像你期望的那样工作。

合并表对性能的影响

MySQL 对合并表的实现对性能有一些重要的影响。和其他 MySQL 特性一样，它在某些条件下性能会更好。下面是关于它的一些注意事项：

- 合并表比含有同样数据的非合并表需要更多的文件描述符。尽管合并表看上去是一个表，它实际是逐个打开了下属表。这样的结果就是单个表的缓存可以创建许多文件描述符。因此，即使已经配置了表的缓存，让服务器线程的文件描述符数量不要超过操作系统的限制，合并表仍然有可能导致超过这一限制。
- 创建合并表的 CREATE 语句不会检查下属表是否是兼容的。如果下属表的定义有轻微的不一样，MySQL 会创建合并表，但是却无法使用。同样，如果在创建了一个有效的合并表之后对某个下属表进行了改变，它也会无法工作，并且会显示下面的错误信息：“ERROR 1168 (HY000)：无法打开定义不同的下属表，或者非 MyISAM 表，或者不存在的表”。
- 访问合并表的查询访问了每一个下属表。这也许会使单行键查找比单个表慢。在合并表中限制下属表是一个好主意，尤其是它是联接中的第二个或以后的表。每次操作访问的数据越少，那么访问每个表的开销相对于整个操作而言就越重要。下面是一些如何使用合并表的注意事项：
 - 范围查找受访问所有下属表的开销的影响小于单个查找。
 - 对索引表的表扫描和对单个表一样快。

- 一旦唯一键和主键查询成功，它们就立即停止。在这种情况下，服务器会挨个访问下属表，一旦查找到了值，就不会再查找更多的表。
- 下属表读取的顺序和 CREATE TABLE 语句中定义的一致。如果经常需要按照特定的顺序取得数据，可以利用这种特性使合并排序操作更快。

合并表的长处

合并表在处理数据方面既有积极的一面，也有消极的一面。经典的例子就是日志记录。日志是只追加的，所以可以每天用一个表。每天创建新的表并把它加入到合并表中。也可以把以前的表从合并表中移除掉，把它转化为压缩的 MyISAM 表，再把它加回到合并表中。

这并不是合并表的唯一用途。它们通常都被用于数据仓库程序，因为它的另一个长处就是管理大量的数据。在实际中不太可能管理一个 TB 级别的表，但是如果是由单个 50GB 的表组成的合并表，任务就会简单很多。

当管理极其巨大的数据库时，考虑的绝不仅仅是常规操作。还要考虑崩溃与恢复。使用小表是很好的主意。检查和修复一系列的小表比起一个大表要快得多，尤其是大表和内存不匹配的时候。还可以并行地检查和修复多个小表。

数据仓库中另外一个顾虑就是如何清理掉老的数据。对巨型表使用 DELETE 语句最佳状况下效率不高，而在最坏情况下则是一场灾难。但是更改合并表的定义是很简单的，可以使用 DROP TABLE 命令删除老的数据。这可以轻易地实现自动化。

合并表并非只对日志和大量数据有效。它可以方便地按需创建繁忙的表。创建和删除合并表的代价是很低的。索引可以像对视图使用 UNION ALL 命令那样使用合并表。但它的开销更低，因为服务器不会把结果放到临时表中然后再传递给客户端。这使得它对于报告和仓库化数据非常有用。例如，要创建一个每晚都会运行的任务，它会把昨天的数据和 8 天前、15 天前、以及之前的每一周的数据进行合并。使用合并表就可以创建无须修改的查询，并且自动地访问合适的数据。甚至还可以创建临时合并表，这是视图无法做到的。

因为合并表没有隐藏下属的 MyISAM 表，所以它提供了一些分区表无法提供的特性：

- 一个 MyISAM 表可以包含很多合并表。
- 可以通过拷贝 .frm、.MYI、.MYD 文件在服务器之间拷贝下属表。
- 可以轻易地把更多的表添加到合并表中。这只需要创建一个新表并且更改合并定义即可。
- 可以创建只包含想要的数据的临时合并表，例如某个特定时间段的数据。这是分区表无法做到的。
- 如果想对某个表进行备份、恢复、更改、修复，或者其他的操作，可以把它从合并表中移除，完成所有的工作之后再把它加回来。
- 可以使用 mysampack 压缩某些或所有的下属表。

分区表正好相反，MySQL 隐藏了分区表的分区，并只能通过分区表访问所有的分区。

5.10.2 分区表

Partitioned Tables

MySQL 分区表的实现在本质上和合并表非常相像。但是，它紧密地和服务器结合在一起，并且和合并表有一个重大的区别：任何一个给定的数据行只会被存储在一个合适的分区上。表的定义基于分区函数（Partitioning

Function), 它约定了行和分区之间的映射关系, 我们稍后再讲解这一点。这意味着主键和唯一键是对整个表起作用的。并且 MySQL 优化器可以更智能地优化分区表。

下面是分区表的一些重要的益处:

- 可以把某些行放在一个分区中, 这可以减少服务器检查数据的数量并且使查询更快。例如, 如果按照日期进行分区, 然后对某个日期范围内数据的查询就可以只访问一个分区。
- 分区数据比非分区数据更好维护, 并且可以通过删除分区来移除老的数据。
- 分区数据可以被分布到不同的物理位置, 这样服务器可以更有效地使用多个硬盘驱动器。

MySQL 对分区的实现还在不停地变化, 它非常复杂, 我们不会在这儿讨论所有的细节。我们讨论主要集中在性能上, 关于基础知识, 最好去查阅 MySQL 手册。最好能把分区那一章通读一遍, 并且仔细了解 CREATE TABLE、SHOW CREATE TABLE、ALTER TABLE、INFORMATION_ SCHEMA、PARTITIONS 和 EXPLAIN。分区使 CREATE TABLE 和 ALTER TABLE 更复杂了。

和合并表一样, 分区表实际也是在存储引擎层由有独立索引的单个的表(分区)组成的。这意味着分区表的内存和文件描述符的要求和合并表类似。但是, 分区不能从表中独立访问, 并且每个分区只能属于一张表。

如同前文所说, MySQL 使用分区函数来决定行到底会被保存到哪个分区中。该函数会返回一个可变的、但是确定的整数。一共有几种分区方式。按范围(Range)分区对每个分区设定了范围值, 然后把行基于其范围放入分区中。MySQL 也支持键(Key)分区、哈希(Hash)分区和列表(List)分区。每种类型都有其优势和劣势, 尤其是在处理主键的时候。

分区为何可以工作

MySQL 设计分区表的一个关键就是把分区看作一个粗糙的索引。假设一个表有 10 亿行对每一天, 每一种物品的销售数据, 并且每一行都比较大, 假设是 500 字节。只会在表中插入数据, 却永远不会更新数据。在大部分情况下, 都是按照日期来检索数据。对该表进行查询的主要问题就是它的大小: 大约 500GB 没有任何索引的数据。

一种加快查询的办法就是在 (day, itemno) 上添加主键, 并且使用 InnoDB。这会把每天的数据物理上绑定在一起, 所以范围查询就可以检索较少的数据。另外, 还可以使用 MyISAM 并且按照想要的顺序插入数据, 这样索引扫描就不会引起大量的随机 I/O 读取。

另外一种选择就是不使用主键, 而把数据按日期进行分区。每次访问某个日期段的数据将会扫描整个分区, 但是这会比在巨型表上使用索引查找要好得多。分区有一点点像索引, 它大致告诉 MySQL 在什么地方去查找数据。但是, 它实际上没有使用内存或磁盘空间, 精确地说来, 这是因为分区没有像索引那样指向具体的行。

但是不要同时加上主键和对表进行分区, 这有可能会降低性能。尤其是要对所有分区进行扫描的时候。在考虑分区时, 要仔细地做性能评测, 因为分区表并不总是能提高性能。

分区示例

下面用两个简短的示例说明分区的好处。首先看看如何设计分区表来存储基于日期的数据。假设已经按照产品对订单和销售数据进行了统计。因为常常需要按照日期范围运行查询, 所以会把订单日期放在主键的第一位, 并且使用 InnoDB 按照日期聚集(Cluster)数据。现在可以对日期进行分区, 进行较高层次的聚集。下面是表

的基本定义，没有使用任何分区策略：

```
CREATE TABLE sales_by_day (
  day DATE NOT NULL,
  product INT NOT NULL,
  sales DECIMAL(10, 2) NOT NULL,
  returns DECIMAL(10, 2) NOT NULL,
  PRIMARY KEY(day, product)
) ENGINE=InnoDB;
```

通常会对基于日期的数据按年或按天进行分区。YEAR()和 TO_DAYS() 函数可以用于数据分区。通常情况下，进行范围分区的好的函数可以在想创建分区的值之间形成线性关系，下面按年进行分区：

```
mysql> ALTER TABLE sales_by_day
-> PARTITION BY RANGE(YEAR(day)) (
-> PARTITION p_2006 VALUES LESS THAN (2007),
-> PARTITION p_2007 VALUES LESS THAN (2008),
-> PARTITION p_2008 VALUES LESS THAN (2009),
-> PARTITION p_catchall VALUES LESS THAN MAXVALUE );
```

现在插入的行会根据 day 的值被放入到合适的分区。

```
mysql> INSERT INTO sales_by_day(day, product, sales, returns) VALUES
-> ('2007-01-15', 19, 50.00, 52.00),
-> ('2008-09-23', 11, 41.00, 42.00);
```

稍后会使用这个例子中的数据。但是在继续之前，我们要在这儿指出一个重要的局限：添加更多的年份会改变表，如果表很大的话，代价会很高（这儿已经假设表很大了，如果不大的话，也没有必要使用分区）。一个比较好的办法就是预先定义好更多的年份，即使在很长时间内你都用到不到这些年份，但是预先把它们包含进来也不会影响性能。

分区表另外一个常用的用途就是分布大表中的行。假设对一个大表运行很多查询。如果想用不同的磁盘为查询服务，那么就会要求 MySQL 将数据行分布到不同的磁盘上。这时不用考虑把相关的数据放在一起，只须简单地把数据平均地分布到磁盘上。下面的例子可以让 MySQL 按照主键的模式分摊数据。它是一种在分区中均匀分布数据的好办法：

```
mysql> ALTER TABLE mydb.very_big_table
-> PARTITION BY KEY(<primary key columns>) (
-> PARTITION p0 DATA DIRECTORY='/data/mydb/big_table_p0/',
-> PARTITION p1 DATA DIRECTORY='/data/mydb/big_table_p1/');
```

可以用磁盘阵列（RAID）控制器实现同样的目标。而且有时候效果会更好，因为磁盘阵列是用硬件执行的，它隐藏了工作的细节，所以不会给数据库的结构和查询带来额外的复杂性。如果目标仅仅是将数据进行物理分布的话，它能提供更好、更一致的性能。

分区表的局限

分区表并不是“银弹”。它现在有如下的局限：

- 当前，所有的分区都要使用同样的存储引擎。例如，不能像合并表那样只压缩部分分区。
- 分区表上的每一个唯一索引必须包含由分区函数引用的列。这样的后果就是很多指导性的示例都避免使用主键。尽管这对于包含没有主键或唯一索引的表的数据仓库而言，是很普遍的现象，但是它对于联机事务处理（OLTP）系统并不常见。相应地，对数据如何分区的选择也会受到限制。

- 尽管 MySQL 能避免分区表的查询访问所有的分区，但是它仍然锁定了所有的分区。
- 分区函数中能使用的函数和表达式有很多限制。
- 一些存储引擎不支持分区。
- 分区不支持外键。
- 不能使用 `LOAD INDEX INTO CACHE`。

还有很多其他的限制（至少在写本书的时候是这样的，MySQL5.1 现在还没有发布）。分区表的灵活程度在某种程度上比合并表要小一些。例如想给分区表添加索引，这个操作并不能在一小段时间内完成，因为 `ALTER` 会将表锁住，并且重新构建整个表。合并表有更多的灵活性，比如可以给一个下属表添加索引。同样地，不能一次只备份或恢复一个分区，但是合并表没有这个限制。

表是否能从分区中得到好处取决于许多因素，应该在应用程序中做实际的评测，以确认它是否是一个好的选择。

利用分区表优化查询

分区引入了一种新的优化查询的方式（当然，也有相应的缺点）。优化器可以使用分区函数修整（Prune）分区，或者把分区从查询中完全移除掉。它通过推断是否可以在特定的分区上找到数据来达成这种优化。因此在最好的情况下，修整可以让查询访问更少的数据。

重要的是要在 `WHERE` 子句中定义分区键，即使它看上去像是多余的。通过分区键，优化器就可以去掉不用的分区，否则的话，执行引擎就会像合并表那样访问表的所有分区，这在大表上会非常慢。

可以使用 `EXPLAIN PARTITIONS` 检查优化器是否去除了分区，还是使用前面例子的数据：

```
mysql> EXPLAIN PARTITIONS SELECT * FROM sales_by_day\G
***** 1. row *****
      id: 1
    select_type: SIMPLE
        table: sales_by_day
    partitions: p_2006,p_2007,p_2008
         type: ALL
possible_keys: NULL
          key: NULL
        key_len: NULL
         ref: NULL
         rows: 3
       Extra:
```

正如所见，查询访问了所有的分区。现在看看在 `WHERE` 子句添加一个约束之后有什么不同：

```
mysql> EXPLAIN PARTITIONS SELECT * FROM sales_by_day WHERE day > '2007-01-01'\G
***** 1. row *****
      id: 1
    select_type: SIMPLE
        table: sales_by_day
    partitions: p_2007,p_2008
```

优化器很聪明，知道如何去除分区。它甚至把范围转换成了一个离散的列表并修整了列表中的每一项。但是，它不是全知全能的，例如，下面的 `WHERE` 子句在理论上是可以修整的，但是 MySQL 却做不到：

```
mysql> EXPLAIN PARTITIONS SELECT * FROM sales_by_day WHERE YEAR(day) = 2007\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: sales_by_day
  partitions: p_2006,p_2007,p_2008
```

在现在的设计中，MySQL 只能通过对分区函数中的列进行修整。它不能修整表达式的结果，即使是表达式和分区函数一样也不行。可以把查询转换成相等的形式，如下：

```
mysql> EXPLAIN PARTITIONS SELECT * FROM sales_by_day
-> WHERE day BETWEEN '2007-01-01' AND '2007-12-31'\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: sales_by_day
  partitions: p_2007
```

WHERE 子句直接引用了分区列，优化器现在就可以使用修整了。

优化器也能在处理查询的过程中修整分区。例如，如果一个分区表联接的是第二个表，并且联接条件是分区键，MySQL 就会在相关的分区中搜索匹配的行。这和合并表很不一样，它总是查找所有的下属表。

5.11 分布式 (XA) 事务

Distributed (XA) Transactions

存储引擎事务在存储引擎内部被赋予了 ACID (译注 1) 属性，分布式 (XA) 事务是一种高层次事务，它可以利用两段提交的方式将 ACID 属性扩展到存储引擎外部，甚至数据库外部。MySQL 5.0 及其以上的版本部分支持 XA 事务。

XA 事务需要事务协调员，它会通知所有的参与者准备提交事务 (阶段一)。当协调员从所有参与者那里收到“就绪 (Ready)”信号时，它会通知所有参与者进行真正的提交 (阶段二)。MySQL 可以是 XA 事务的参与者，但不能是协调员。

MySQL 内部其实有两种 XA 事务。MySQL 服务器能参与由外部管理的分布式事务，但它内部使用了 XA 事务来协调存储引擎和二进制日志。

5.11.1 内部 XA 事务

Internal XA Transactions

MySQL 内部使用 XA 事务的原因是服务器和存储引擎之间是隔离的。存储引擎之间是完全独立的，彼此不知道对方的存在，所以任何跨引擎的事务本质上都是分布的，并且要求第三方来进行协调。MySQL 就是第三方。假如没有 XA 事务，跨引擎事务提交需要顺序地要求每个引擎进行提交。这样就会引入一种可能，就是在某个引擎提交之后发生了崩溃，但是另外一个引擎还未提交。这就打破了事务的原则。

如果把记录事件的二进制日志看成一个“存储引擎”，那么就能理解为什么即使是单个事务性引擎也需要 XA 事

译注 1: ACID, 是指在数据库管理系统 (DBMS) 中事务所具有四个特性: 原子性 (Atomicity)、一致性 (Consistency)、隔离性 (Isolation, 又称独立性)、持久性 (Durability)。

务。存储引擎把事件提交给二进制日志时，需要和服务器进行同步，因为是服务器，而不是存储引擎处理二进制日志。

当前的 XA 在性能上有些进退两难。它打破了 InnoDB 从 MySQL 5.0 以来的对群体提交（Group Commit）（一种使用单次 I/O 提交多个事务的技术）的支持，所以会导致了很多 `fsync()` 调用。如果二进制日志处于激活状态，那么每个事务都会需要等待日志同步，并且每次事务提交都要求两次日志重写，而不是一次。换句话说，如果想让事务和二进制日志安全地同步，就会要求至少三次 `fsync()` 调用。防止其发生的唯一办法就是禁用二进制日志并把 `innodb_support_xa` 设置为 0。

这样设置无法兼容复制。复制需要二进制日志和 XA 支持，并且为了尽可能地安全，还须要把 `sync_binlog` 设置成 1，这样设置就能对存储引擎和二进制日志进行同步。（否则的话，XA 支持就没有必要了，因为二进制日志不会被提交到磁盘上）。这是强烈推荐使用带有备用电池的写入缓存的磁盘阵列控制器的一个原因，它能加快 `fsync()` 调用并且恢复性能。

下一章将会详细讲解如何配置事务日志及二进制日志。

264

5.11.2 外部 XA 事务

External XA Transactions

MySQL 可以参与，但不能管理外部分布式事务。它不支持完整的 XA 规范。例如，XA 规范允许连接运行单个事务中的连接，但是 MySQL 现在还不能做到这一点。

外部 XA 事务的开销比内部 XA 事务更高，这是因为延迟会增加，并且参与者失败的可能性更大。在 WAN、甚至是因特网上使用 XA，一个常见的问题就是网络性能不可预测。当有不可预测的部分，比如较慢的网络或一个有可能很久都不点击“保存”按钮的用户，最好的选择就是避免 XA 事务。任何耽搁提交的因素都会有很高的代价，因为它导致的不是单个系统延迟，而是许多系统。

可以用另外的方式设计分布式事务。例如，可以在本地把数据插入队列，然后把它自动地分布成小而快的事务。也可以使用 MySQL 复制把数据从一个地方搬运到另外一个地方。我们也发现某些使用了分布式事务的应用程序其实根本没必要使用事务。

总的说来，XA 事务是一种在服务器之间同步数据的有用的方式。如果因为某些原因，比如不能使用复制或数据更新的性能并不是关键因素，它的效果会不错。

优化服务器设置

Optimizing Server Settings

人们通常会问：“我的服务器有 16GB 内存，100GB 数据，需要怎样的优化配置文件？”这样的文件其实并不存在。具体的配置文件依赖于服务器的硬件、数据量、查询类型、响应时间、事务持久性和连续性等因素。

MySQL 的默认配置不适用于使用大量资源，因为其通用性很高，通常不会假设机器上只安装 MySQL。在默认情况下，配置文件只够启动 MySQL 并且对少量的数据运行简单的查询。如果有较多的数据，那么肯定需要对配置文件进行定制。可以先尝试使用随 MySQL 发布的配置文件，然后按照需要对它进行配置。

不要期望改变配置文件会带来巨大的性能提升。提升的具体大小取决于工作负载，通常可以通过选择适当的配置参数得到两到三倍的性能提升（具体什么参数起作用取决于多种因素）。在这之后，性能提升就是增量的。通过改变一两个配置参数，您可以看到某些原本运行较慢的查询快了一些，但是，服务器的性能通常不会提升一个数量级。为了达到更大的提升，通常需要检查服务器架构、查询及应用程序的架构。

本章先说明了 MySQL 的配置选项如何起作用并且该如何更改它们。接下来讨论了 MySQL 如何使用内存及如何优化内存使用。然后以同样的详细程度讨论了 I/O 及磁盘存储。接下来是基于工作负载的调优，这有助于在特定的负载下得到最佳性能。最后讨论了对某些查询进行动态参数设置的方法。



提示：关于术语的说明：因为许多 MySQL 的命令行选项都和服务器变量相对应，所以有时不会对它们进行区分。

266

6.1 配置基础知识

Configuration Basics

本节概述了如何成功地配置 MySQL。我们首先解释了 MySQL 配置实际是如何工作的，然后讨论一些最佳实践。MySQL 通常对配置相当宽容，但是接受这些建议也许能节约您大量的时间和工作。

首先要知道的是 MySQL 从什么地方得到配置信息：命令参数和配置文件中的设置。在 Unix 类系统中，配置文件通常位于 `etc/my.cnf` 或 `etc/mysql/my.cnf`。如果使用操作系统的启动脚本启动 MySQL，这通常是唯一一个配置选项的地方。如果手动启动 MySQL，也可以通过命令行定义参数。



提示：大部分变量和相应的命令行选项有同样的命令，但是也有一些例外。比如，`-memlock` 设置了变量 `locked_in_memory`。

任何长久使用的设置都应该被放到全局配置文件中，而不是通过命令行进行定义。否则就有可能忘记使用它们。把所有的文件放在一个地方也是一个好主意，这样的话就可以很容易地检查它们。

要确认自己知道服务器的配置文件在什么地方。我们曾经见过有人用 MySQL 根本无法读取的文件进行调优。比如在 Debian GNU/Linux 机器上, 实际起作用的是/etc/mysql/my.cnf, 而不是/etc/my.cnf。有时在好几个地方都有文件, 这也许是因为前一位系统管理员也被弄糊涂了。如果不知道服务器读取的是哪个文件, 可以进行查询:

```
$ which mysqld
/usr/sbin/mysqld
$ /usr/sbin/mysqld --verbose --help | grep -A 1 'Default options'
Default options are read from the following files in the given order:
/etc/mysql/my.cnf ~/.my.cnf /usr/etc/my.cnf
```

这适用于典型安装, 即主机上只有一个服务器的情况。可以设计更复杂的配置, 但是并没有标准的办法。MySQL 包含了一个叫 mysqlmanager 的程序, 它可以使用同一个文件的不同部分运行出多个配置文件的实例 (它是较老的 mysqld_multi 脚本的替代品)。但是, 许多操作系统没有在启动脚本里面包含这个程序。事实上, 许多人根本就不会使用 MySQL 提供的启动脚本。

配置文件被分成了若干部分, 每部分的第一行都是用方括号括起来的该部分的名字。MySQL 程序会读取和程序名同名的部分, 并且许多客户端程序会读取 client 部分, 这儿也是放置通用设置的地方。服务器通常会读取 mysqld 部分。要确保自己的设置被放在了正确的部分里面, 否则它们就不会有任何作用。

6.1.1 语法、作用域及动态性

Syntax, Scope, and Dynamism

配置设置都是小写的, 使用下划线或破折号分割单词。下面的两个设置是一样的, 在命令行和配置文件中都可使用这两种格式:

```
/usr/sbin/mysqld --auto-increment-offset=5
/usr/sbin/mysqld --auto_increment_offset=5
```

我们的建议是选择并坚持一种格式。这会让搜索设置容易得多。

配置设置有几种作用域。一些设置在整个服务器内都有效 (全局域); 另外一些针对每个连接 (会话域); 还有一些只对对象有效。许多会话域的变量和全局变量是一样的, 可以认为是全局变量提供了默认值。如果修改了会话域变量的值, 它只会在当前连接内有效, 连接关闭后值就消失了。下面是一些值得注意的例子:

- query_cache_size 变量是全局性的。
- sort_buffer_size 变量有全局性的默认值, 但是可以在会话中进行设置。
- join_buffer_size 有全局性的默认值, 并且可以在会话中进行设置。但是联接了几个表的查询可能会为每一个联接都分配一个联接缓冲区, 所以每个查询都可能有几个联接缓冲区。

除了在配置文件中设置变量, 也可以在服务器运行的时候对某些值 (不是全部值) 进行设置。MySQL 把它们叫做动态变量。下面的语句显示了几种不同的设置会话和全局变量的方法:

```
SET                sort_buffer_size = <value>;
SET GLOBAL         sort_buffer_size = <value>;
SET                @@sort_buffer_size := <value>;
SET @@session.sort_buffer_size := <value>;
SET @@global.sort_buffer_size := <value>;
```

如果动态地设置了变量，那么它们在 MySQL 关闭之后就会丢失。如果想保留这些设置，就应该同时更新配置文件。

如果在服务器正在运行的时候设置了一个全局变量的值，当前会话和其他已有会话的值不会受到影响。这是因为会话的值是在连接创建的时候从全局变量初始化的。可以在每次更改值后使用 `SHOW GLOBAL VARIABLES` 检查一下结果，确保更改已经生效。

变量使用不同的单位，应该要知道每个变量正确的单位是什么。比如，`table_cache` 变量定义了能被缓存的表的数量，而不是被缓存的字节数。`key_buffer_size` 的单位是字节，然而还有其他变量的单位是页面或者其他单位，例如百分比。

许多变量都可以使用前缀，比如 `1MB` 指 1 兆字节。然而，这只在配置文件或命令行参数中有用。当使用 `SET` 命令时，必须使用字面值 `1 048 576` 或者 `1024×1024`。在配置文件中无法使用表达式。

还可以使用 `SET` 给变量赋一个特殊的值：`DEFAULT`。给会话域内的变量赋 `DEFAULT` 值将会使其变成相应的全局变量的值；给一个全局变量赋 `DEFAULT` 将会使其得到编译时的默认值（不是配置文件中的值）。这对重置会话变量的值非常有用。我们建议不要对全局变量使用这个值，因为这有可能和期望的情况不太一样，那是因为它不会将变量设置成服务器刚启动时的值。

6.1.2 设置变量的副作用

6.1.2.1 Effects of Setting Variables

动态设置变量有出人意料的副作用，比如会清空缓冲区。要注意在线更改的设置，因为它可能会导致服务器做大量的工作。

有时从变量的名字推断变量的行为。比如，`max_heap_table_size` 做的事情和名字一样：定义了隐式的内存临时表能增长到的最大值。但是，命名规范并不完全一致，所以不能总是从名字推测变量的行为。

下面来看看某些重要的变量及动态地改变它们造成的影响：

`key_buffer_size`

设置这个变量给键缓冲区（或者说键缓存）分配指定大小的空间。但是操作系统只有在实际用到这些空间的时候才会进行分配。例如，将键缓冲区大小设置为 1GB，并不意味着服务器就会真正地给它分配 1GB 空间（在下一章就会讨论如何观察服务器内存使用情况）。

可以创建多个键缓存，本章稍后将会谈到这个问题。如果对于一个非默认大小的键缓存，将它的值设置为 0，MySQL 就会把每一个索引从特定的缓存移到默认的缓存中，并且在没有对象使用特定的缓存时，就会将其删除掉。给一个不存在的缓存设置这个变量将会创建缓存。

对一个已有的缓存设置非零值将会冲洗缓存，从技术上来说，这是一个在线操作，但是它会阻止所有访问该缓存的动作，直到缓存冲洗完成。

`table_cache_size`

设置这个变量不会立即生效，要等到下一个线程打开表的时候才会生效。当它生效的时候，MySQL 会检查变量的值。如果值大于缓存中表的数量，线程就可以把新打开的表插入到缓存中。如果值小于缓存中表

的数量，MySQL 就会从缓存中删除掉没有使用的表。

`thread_cache_size`

设置这个变量不会立即生效，生效被延迟到了下一次线程关闭的时候。在那时，MySQL 检查缓存中是否有空间存储线程。如果是，它会把线程缓存起来，供另外一个连接使用。如果不是，它会直接结束掉线程。在这种情况下，缓存中线程的数量，以及线程缓存使用的内存数量不会立即就下降。只有当新连接为了使用线程把它从缓存中移走的时候才会看到下降。（MySQL 只有在连接关闭的时候才会把线程加入缓存，也只有在创建新连接的时候才从缓存中移除线程。）

`query_cache_size`

在服务器启动的时候，MySQL 会为查询缓存一次性分配变量所定义数量的内存。如果更新了变量（即使是把值设置为当前值），MySQL 会立即删除掉所有缓存的查询，重新把缓存设置为定义的大小，并且重新初始化缓存的内存。

`read_buffer_size`

MySQL 只有在查询需要的时候才会为该缓存分配内存，但是它会一次性地把指定的大小分配给该缓存。

270 `read_rnd_buffer_size`

MySQL 只在查询需要的时候才会给这个缓冲区分配内存，并且只会分配所需的内存（该变量更精确的名字应该是 `max_read_rnd_buffer_size`）。

`sort_buffer_size`

MySQL 只有在查询需要排序的时候才会为这个缓冲区分配内存。但是只要发生了排序，MySQL 会立即分配变量定义的所有内存，不管是否需要这么大的空间。

在本书另外的地方对这些变量有更详细的解释。这儿的目的是简单地展示更改这些重要的变量会发生什么事情。

6.1.3 开始配置

Getting Started

设置变量的时候要小心。更大的值并不总是好事情，如果将值设得太高，很容易引发诸多问题：耗尽内存、导致服务器使用交换区、耗尽地址空间等。

我们建议你在运行服务器之前就开发一套基准测试组件（第 2 章已经讨论过基准测试）。为了优化服务器配置，需要用测试组件来模拟总体工作负载和一些边界条件，比如极其庞大和复杂的查询。如果已经确认了一个特定的问题，比如某个查询运行很慢，那么就可以试着优化这个查询，但是这有可能会在你毫不知情的情况下为其他查询带来负面影响。

应该总是使用监控系统来衡量改动是提升了还是损害了服务器总体性能。基准测试是不够的，因为它们覆盖的范围不够广。如果不衡量服务器的总体性能，就有可能在不知情的情况下伤害性能。我们曾经见过很多例子，某些人改动了服务器的配置，以为可以提高性能，但由于工作负载的变化，实际上却损害了服务器的总体性能。第 14 章讨论了一些监控系统。

最好的方式每次只小幅度地改动一两个设置，并且在每次改动之后都进行基准测试。有时结果会让你惊讶，也许增加一点点就带来了性能提升，但是再增加一点就让性能急剧下降。如果在某次改动后性能下降了，那么有可能是过多地请求了某种资源，比如为某个经常被分配和释放的缓冲区请求了过多内存。也有可能是在服务器和操作系统或硬件之间造成了某种不匹配。例如，`sort_buffer_size` 有可能会受 CPU 缓存的影响，`read_buffer_size` 要和服务器的预读和通用 I/O 子系统相匹配。更大的值并不总是意味着更好。一些变量也会

依赖于其他变量。了解这些东西需要经验，也需要对系统架构有一定的了解。比如，`innodb_log_file_size` 的最佳大小就依赖于 `innodb_buffer_pool_size`。

如果习惯做笔记，比如在配置文件中添加注释，这会为你（以及你的继任者）节约大量的工作。一个更好的办法就是对配置文件进行版本控制。这实际上非常有好处，因为可以回滚自己的改动。为了减少管理大量配置文件的复杂性，可以简单地在配置文件中创建一个到中央版本控制库的符号链接。在一些系统管理的书籍中有很多这方面的内容。

在对配置进行调优之前，应该对查询和结构进行调优，进行一些最基本的优化，比如添加索引。如果已经对配置文件做了很多调整，再回过头来更改查询和架构，那么就有可能要重新调整配置文件。要知道调优是一个渐进的过程。除非硬件、工作负载及数据是完全静态的，否则就需要在随后的工作中对配置进行再次调整。这意味着并不需要一次性把服务器的性能调到最好，实际上，对配置文件花费大量的时间也许会收效甚微。我们的建议就是让配置保持“够好”就行了。除非忘记了某项重要的设置，否则就不需要再次改动它。当更改了查询或架构的时候，就可以回过头来再次修改配置文件。

我们通常会为不同的目的开发不同的配置文件样本，然后把它们当成默认设置。这一点在安装大量类似的服务器时尤其有用。但是，正如本章开头所说的那样，并没有一个通用的配置文件适合所有服务器。每个人都需要开发自己的配置文件，因为好的起点依赖于如何使用服务器。

6.2 通用调优原则

General Tuning

可以把调整配置文件看成是一个两步的过程：在安装的时候使用适当的初始值，然后基于工作负载进行细节调整。

你也许会使用 MySQL 提供的样本配置作为起始点。考虑服务器的硬件会帮助做出选择。服务器有多少内存、硬盘和 CPU？那些样本文件的名字通常是一目了然的，比如 `my-huge.cnf`、`my-larege.cnf`、`my-small.cnf`。但是，这些样本文件通常只适用于 MyISAM 表。如果使用了其他的存储引擎，就需要创建自己的配置文件。

6.2.1 内存使用调优

Tuning Memory Usage

配置 MySQL 正确地使用内存对性能至关重要。对 MySQL 内存使用进行定制基本上是肯定的。可以认为 MySQL 的内存消耗有两种范畴：可以控制的和不可控制的。你不能控制 MySQL 使用多少内存来运行服务器、解析查询及管理内部运行，但是可以控制它为特定工作使用多少内存。妥善使用可控内存并不是难事，但是需要你明白自己配置的是什么。

可以用下面的方式进行内存调优：

1. 决定 MySQL 能使用的内存的绝对上限。
2. 决定 MySQL 会为每个连接使用多少内存，比如排序缓冲区和临时表。
3. 决定操作系统需要多少内存来很好地运行自身，包括机器上的其他程序，比如周期性工作。
4. 假设上面的工作都已完成，就可以把剩余的内存分配给 MySQL 的缓存，比如 InnoDB 的缓存池。

下面几节会依次解释上面的步骤，然后会对 MySQL 不同的缓存的要求做详细讲解。

MySQL 能使用多少内存

在特定的系统上，MySQL 可能使用的内存有一个绝对的上限。起始点是服务器物理内存的数量。如果服务器没有，那 MySQL 肯定就用不了。

也可以考虑操作系统或系统架构的限制，比如 32 位系统能给每一个进程分配的最大内存。MySQL 是一个有多个线程的单进程程序，所以它能使用的内存数量一定会受操作系统的限制。比如，在 32 位 Linux 内核的操作系统上，单个进程能使用的地址空间在 2.5GB 到 2.7GB 之间。耗尽地址空间是很危险的，会导致 MySQL 崩溃。

还有很多其他和操作系统相关的参数也需要考虑进来，不仅仅包括单个进程的限制，还要考虑堆栈大小和其他设置。系统函数库也会对分配造成影响，比如函数库不能一次分配 2GB 以上的内存，那么就不能把 `innodb_buffer_pool` 的值设得比 2GB 高。

273 即使是在 64 位系统上，一些限制仍然存在。例如，许多我们讨论的缓冲区，比如键缓冲，在 64 位系统上被限制在 4GB。一些限制在 MySQL 5.1 中被放开了，并且以后放松的空间可能会更大，因为 MySQL AB 正在让 MySQL 使用更加强大的硬件。MySQL 手册规定了每个变量的最大值。

单个连接需要的内存

MySQL 只需要很少的内存保持连接开启。它也需要一定的基本内存来执行查询。你需要在 MySQL 工作负载处于顶峰的时候为它分配足够的内存，否则查询就会因为内存不足而变得很慢，甚至失败。

知道 MySQL 在负载处于顶峰的时候需要多少内存是有用的，但是某些使用模式会出人意料地消耗大量的内存，这使得内存消耗变得难以预料。准备语句是一个例子，因为可以一次性打开很多准备语句。另外一个例子就是 InnoDB 的表缓存（稍后有更多这方面的内容）。

在预测最高内存消耗的时候不用假设最坏的场景。例如，如果 MySQL 被配置成可以接受 100 个连接，理论上应该可以同时在这 100 个连接里面运行大的查询，但是实际上这不可能发生。例如，如果把 `myisam_buffer_size` 设置为 256MB，最坏情况下就会消耗 25GB 内存，但是如此高的消耗在实际中不太可能发生。

和计算最坏情况相比，比较好的办法就是观察服务器在真实负载下的内存消耗，可以通过观察进程虚拟内存大小得到相应的数据。在许多 Unix 系统中，可以在 `top` 的 `VIRT` 栏，或者是 `ps` 的 `VSZ` 栏看到这一数据。下章会对内存检测做更多说明。

为操作系统保留内存

和为查询指定内存一样，还应当为操作系统保留足够的内存。这样最大的好处就是服务器不会主动把虚拟内存

保存在磁盘上（更多详情请查看第 334 页的“交换”）。

不需要为操作系统保留多于 1GB 或 2GB 的内存，即使服务器安装了很多内存也不需要。添加一点是为了保险，如果机器上有消耗内存的周期性任务（比如备份），那么还可以多给一点。不要为操作系统的缓存添加任何内存，因为这些缓存可能会很大。操作系统通常会为这些缓存使用剩余的内存，并且我们通常认为它们独立于操作系统自身需要的内存。

为缓存分配内存

如果服务器是 MySQL 专用的，就不需要为操作系统或用于处理查询的缓存保留任何内存。

MySQL 缓存比其他的東西需要更多的内存。它使用缓存来避免磁盘访问。磁盘访问比内存访问慢几个数量级。操作系统也许会为 MySQL 缓存一些数据（尤其是为 MyISAM），但是 MySQL 自己也需要大量的内存。

对于大部分用户来说，下面的这些缓存是最重要的：

- 操作系统为 MyISAM 的数据提供的缓存。
- MyISAM 键缓存。
- InnoDB 缓存池。
- 查询缓存。

还有另外一些缓存，但是它们通常不会使用太多内存。前一章详细地讨论了查询缓存，所以在接下来的章节将会集中讨论 MyISAM 和 InnoDB 需要的缓存。

如果只使用一个存储引擎，服务器调优就容易得多。如果只使用 MyISAM 表，就可以完全地禁止 InnoDB。如果只使用 InnoDB，就可以只给 MyISAM 分配最少的资源（MySQL 某些内部操作需要 MyISAM）。但是如果混合使用了多个存储引擎，那么就很难在它们之间找到平衡。我们发现最好的方式就是进行有根据的猜测，并且进行基准测试。

6.2.2 MyISAM 键缓存

MyISAM Key Cache

MyISAM 键缓存也被叫做**键缓冲区**。默认只有一个缓冲区，但是可以创建多个。和 InnoDB 及其他存储引擎不同的是，MyISAM 自身只缓存了索引，没有数据（它让操作系统缓存数据）。如果主要是使用 MyISAM，那么就应该为键缓存分配很多内存。



提示：本节的大部分建议都是假设你只使用 MyISAM。如果混合使用了 MyISAM 和其他存储引擎，比如 InnoDB，就需要考虑所有存储引擎的需要。

最重要的选项是 `key_buffer_size`，它的值应该占到所有保留内存的 25% 到 50%。值得一提的是操作系统缓存，它通常用来保存从 MyISAM 的 .MYD 文件中读取出来的数据。对于 MySQL 5.0，不管采用的是什架构，该变量的最大上限都是 4GB。

275 MySQL 5.1 允许更大的值。请查询文档以确定 MySQL 的版本。

在默认情况下, MyISAM 将所有索引缓存在默认的键缓冲中, 但是可以创建多个命名键缓冲区。这样就可以一次在内存中保存 4GB 以上的索引。为了创建名为 `key_buffer_1` 和 `key_buffer_2`, 大小都为 1GB 的键缓冲区, 可以在配置文件中加入下面两行:

```
key_buffer_1.key_buffer_size = 1G
key_buffer_2.key_buffer_size = 1G
```

现在就有 3 个缓冲区了。两个显式创建的和一个默认的。可以使用 `CACHE INDEX` 命令把表映射到缓存。也可以通过下面的命令告诉 MySQL 把表 `t1` 和 `t2` 的索引保存到 `key_buffer_1`:

```
mysql> CACHE INDEX t1, t2 IN key_buffer_1;
```

现在当 MySQL 从这些表的索引中读取数据的时候, 它就会把数据保存到特定的缓冲区了。还可以使用 `LOAD INDEX` 把表的索引预加载到缓存中:

```
mysql> LOAD INDEX INTO CACHE t1, t2;
```

可以把这个 SQL 语句放到一个 MySQL 启动时执行的文件里面。文件名字必须在 `init_file` 选项中定义, 并且文件可以包含多个 SQL 命令, 每个命令一行, 而且不能有注释。MySQL 第一次访问 MYI 文件时, 任何没有显式地映射到键缓冲区的索引都会被放在默认缓冲区中。

可以使用 `SHOW STATUS` 和 `SHOW VARIABLES` 监视键缓冲区的使用情况和性能。可以通过下面的公式检查缓存命中率 and 缓冲区使用的百分比:

缓存命中率

$$100 - ((\text{Key_reads} * 100) / \text{Key_read_requests})$$

缓存使用百分比

$$100 - ((\text{Key_blocks_unused} * \text{key_cache_block_size}) * 100 / \text{key_buffer_size})$$


提示: 介绍了一些工具, 比如 `innotop`, 可以使性能监视更加方便。

了解缓存命中率是很好的, 但是它也可能造成误导。例如, 99% 和 99.9% 之间的差别看上去很小, 但是它实际代表了 10 倍的提升。缓存命中率也依赖于应用程序: 一些程序也许只需要 95%, 但是另外一些 I/O 密集的程序也许需要 99.9%。使用大小适当的缓存, 命中率甚至可以达到 99.99%。

276 每秒内缓存未命中的数量更有实际意义。假设某个硬盘每秒能执行 100 次随机读取, 如果有 5 次未命中不会导致工作成为 I/O 密集型, 但是如果每秒 80 次, 那就会造成问题。可以使用下面的公式计算它的值:

$$\text{Key_reads} / \text{Uptime}$$

时间间隔从 10 秒到 100 秒依次增加, 计算未命中的次数, 就可以了解当前的性能。下面的命令以 10 秒的幅度递增, 计算了总的未命中次数:

```
$ mysqladmin extended-status -r -i 10 | grep Key_reads
```

在决定给键缓存分配多少内存的时候, 知道 MyISAM 索引使用了多少磁盘空间会比较有帮助。没有必要让键缓

存大于数据的大小。如果使用 Unix 系统，可以使用下面的命令找出存储索引的文件大小：

```
$ du -sch `find /path/to/mysql/data/directory/ -name "*.MYI"`
```

要记住 MyISAM 使用操作系统缓存数据文件，它通常大于索引的大小。因此，留给操作系统缓存的内存比留给键缓存的要多也是很正常的。最后，即使没有使用 MyISAM 表，也要给 `key_buffer_size` 设置少量的内存，比如 32MB。MySQL 有时会使用 MyISAM 执行一些内部操作，比如给有 `GROUP BY` 的查询提供临时表。

MyISAM 数据块大小

键数据块大小是重要的（尤其是对于写入密集型的工作负载），因为它导致了 MyISAM、操作系统缓存，以及文件系统之间的交互。如果键数据块太小，就会遇到写入排队的情况，也就是操作系统只有等读取完成之后才能写入。下面是写入排队的例子，假设操作系统的页面大小是 4KB（这是 X86 架构的典型大小）并键数据块的大小是 1KB：

1. MyISAM 从磁盘请求 1KB 数据。
2. 操作系统从磁盘读取 4KB 数据，并缓存起来，然后把需要的 1KB 数据传给 MyISAM。
3. 操作系统将缓存中的数据丢掉。
4. MyISAM 修改了那 1KB 数据并要求操作系统把它写回磁盘。
5. 操作系统从磁盘读取同样的 4KB 数据，放入缓存中，修改其中的 1KB，并且把这个数据块写回磁盘。

写入等待发生在第 5 步，也就是 MyISAM 要求操作系统只写入其中一部分数据的时候。如果 MyISAM 数据块的大小和操作系统读取的页面大小相同，第 5 步就可以避免（注 1）。 277

不幸的是，MySQL 5.0 及其早期版本没有办法配置键数据块的大小。但是，在 MySQL 5.1 及更高版本中，可以使键数据块大小和操作系统相匹配，以避免写入等待。`myisam_block_size` 变量控制了键缓存块的大小。也可以在 `CREATE TABLE` 或 `CREATE INDEX` 语句中为每一个键定义 `KEY_BLOCK_SIZE` 选项来控制键的大小。但是由于所有的键都存储在相同的文件中，所以的确需要它们的大小都等于或大于操作系统的值，以避免由于对齐导致的写入等待问题。（比如，一个键数据块大小是 1KB，另外一个 是 4KB，4KB 大小可能就无法和系统的页面相匹配。）

6.2.3 InnoDB 缓冲池

The InnoDB Buffer Pool

如果主要是使用 InnoDB，InnoDB 缓冲池也许会比其他的東西需要更多内存。和 MyISAM 键缓存不同，InnoDB 缓冲池不仅仅保存了索引，它还保存了行数据及自适应的哈希索引（查看第 101 页的“哈希索引”）、插入缓冲区、锁及其他的内部结构。InnoDB 也使用了缓冲池帮助延迟写入，这样它就可以合并更多的写入然后顺序地执行它们。简言之，InnoDB 严重依赖于缓冲池，并且应该给它分配足够的内存。MySQL 手册建议在专用服务器上把 80% 的物理内存分配给缓冲池。实际上，如果机器有许多内存的话，还可以给它分配更多的内存。和 MyISAM

注 1：理论上，可以把原始的 4KB 数据保存在操作系统的缓存中，从而不需要读取。但是，我们实际上不能控制操作系统把什么数据留在缓存中。可以使用 `fincore` 工具来查看哪个数据块被留在了内存中。可以到 <http://net.doit.wisc.edu/~plonka/fincore/> 下载该工具。

键缓冲区一样，可以使用 SHOW 命令或 innotop 工具来监视 InnoDB 缓冲池的性能和内存使用情况。

InnoDB 没有和 LOAD INDEX INTO CACHE 等价的命令。但是，如果正在给服务器热身，使它为繁重的负载做好准备的话，可以使用查询进行全表扫描或全索引扫描。

在大部分情况下，应该使 InnoDB 缓冲池和可用内存保持一致。但是，在少数情况下，很大的缓冲池（比如 50GB）会导致长时间的延迟。比如，大型的缓冲池在检查点或插入缓存合并操作的时候会变慢，并且并发也会因为锁定而减少。如果遇到了这些问题，就应该减少缓冲池的大小。

278 可以改变 innodb_max_dirty_pages_pct 的值，让 InnoDB 改变保留在缓冲池中被修改的页面的数量。如果允许保留更多被修改的页面，InnoDB 就需要更长的时间来关闭，因为它会在关闭之前把修改的页面写入数据文件。可以强制它快速关闭，但是它在重新启动的时候就会要做更多的恢复工作，所以这实际上不能加快从关闭到启动的周期。如果预先知道需要关闭，就可以把这个变量设置为较小的值，并等待它冲刷线程以清理缓冲池，然后在修改的页面数量变小的时候关闭 InnoDB。可以通过观察状态变量 Innodb_buffer_pool_pages_dirty 或使用 innotop 监视 SHOW INNODB STATUS 的值来监测修改过的页面的数量。

降低 innodb_max_dirty_pages_pct 的值其实并不会确保 InnoDB 会在缓冲池里面保存较少的修改的页面。相反，它控制了 InnoDB 什么时候开始“变懒”。InnoDB 默认的行为是使用后台线程冲刷修改过的页面、合并写入查询并且顺序地执行写入动作。这种行为是“懒惰的”，因为它只会在其他数据需要空间的时候才会执行冲刷行为。当修改过的页面超过变量规定的值，InnoDB 会尽快地冲刷缓冲池，以保持尽可能少的修改过的页面。这个变量的默认值是 90%，所以在默认情况下，InnoDB 只会在修改过的页面已经占据了缓冲池 90%空间的时候，才开始冲刷动作。

如果希望改变写入状况，就可以按照自己的工作负载调整这个值。比如，将它降低到 50%就会让 InnoDB 执行更多的写入操作，因为它会更快地冲刷页面，并且也不会执行批次写入。但是，如果工作负载有很多写入的尖峰，使用更低的值有助于 InnoDB 更好地吸收尖峰。它会有更多的空余内存来保存修改后的页面，所以它不会等待其他修改过的页面被冲刷到磁盘上。

6.2.4 线程缓存

The Thread Cache

线程缓存保存了和当前连接无关的线程。这些线程可以供新连接使用。当一个新连接被创建出来并且缓存中有一个线程的时候，MySQL 就会把这个线程从缓存中删除，并且把它赋给连接。连接关闭时，MySQL 会回收线程，把它放回到缓存中。如果缓存中没空间了，MySQL 就会销毁该线程。只要缓存中有自由的线程，MySQL 就能很快地响应连接请求，因为它不需要为每个连接都创建新的线程。

thread_cache_size 定义了 MySQL 能在缓存中保存的线程数量。除非服务器有很多连接请求，否则就不需要改变这个变量的值。可以通过观察 threads_created 变量的值，以确定线程缓存是否足够大。如果每秒创建的线程数量少于 10 个，缓存的大小就是足够的。但是要让每秒创建的线程少于 1 个，也是很容易的。

279 一个好办法就是观察 Threads_connected 的值，并且把 thread_cache_size 的值设得足够大，以处理波动的负载。例如，如果 Threads_connected 通常在 100 到 200 之间变化，那么就可以把线程缓存设成 100。如果它在 500 到 700 之间变化，将线程缓存设置为 200 就足够了。要这样思考这个问题：有 700 个连接的时候，缓存中也许根本就没有线程。那么当有 500 个连接的时候，缓存中就有 200 个线程了。

对于大多数情况而言，非常巨大的线程缓存是没有必要的。但是让它很小并不会节约内存，所以那么做没什么

好处。每个在缓存中的线程通常消耗 256KB 内存。这和线程处于活动状态时处理查询所需要的内存相比简直微不足道。通常说来，需要把线程缓存保持得足够大，以使 `Threads_created` 不会经常增加。但是如果它的值非常大（比如好几千），那么就应该把它设置得小一点。这是因为操作系统不能很好地处理极其多的线程，即使它们处于睡眠状态也不行。

6.2.5 表缓存

表缓存和线程缓存是相似的概念。但是它存储了能表示表的对象。缓存中的每个对象都包含了解析表后生成的 .frm 文件和其他数据，对象中其他的东西依赖于表的存储引擎。例如，对于 MyISAM 而言，它保存了表的数据或索引文件描述符。对合并表，它保存了大量的描述符，因为合并表有许多下属表。

表缓存有助于复用资源。例如，当查询要求访问 MyISAM 表的时候，MySQL 就可以从缓存中取出一个文件描述符，而不是打开一个文件。表缓存也有助于避免索引头中的 I/O 请求使 MyISAM 表的状态变为“正在使用”（注 2）。

表缓存的设计有一点以 MyISAM 为中心。由于一些历史原因，它属于服务器和存储引擎之间的隔离不是很清晰的区域。对于 InnoDB 来说，表缓存不是那么重要，因为 InnoDB 在很多方面都不会依赖于它（例如保存文件描述符，InnoDB 为此有自己的表缓存）。但是，InnoDB 还是可以从解析后的 .frm 文件中获益。

在 MySQL 5.1 中，表缓存被分为了两个部分：一部分为打开表，另一部分为表的定义（通过 `table_open_cache` 和 `table_definition_cache` 定义）。因此，表的定义（解析过的 .frm 文件）和其他资源是隔离的，比如文件描述符。打开的表仍然基于每个线程、每个使用的表。但是表定义是全局的，可以在所有的连接中共享。通常可以把 `table_definition_cache` 的值设置得足够高，以缓存所有表的定义。除非有上万的表，否则这项工作很容易的。

如果 `Opened_tables` 的值很大或者正在上升，那就说明表缓存不够大，应该增加系统变量 `table_cache` 的值（在 MySQL 中是 `table_open_cache`）。将表缓存变得很大的唯一坏处就是在有很多 MyISAM 表的时候，它会导致较长的关闭时间，因为要冲刷键数据块，而且表要被标记为不再打开。出于同样的原因，它也会导致 `FLUSH TABLES WITH READ LOCK` 需要较长时间才能完成。

如果收到 MySQL 不能打开更多文件的错误提示（使用 `percona-toolkit` 工具检查错误码的详情），那就应该增加 MySQL 允许保持开启的文件数量。可以在 `my.cnf` 文件中设定 `open_files_limit` 解决这个问题。

线程和表缓存其实都不会使用太多内存。它们的好处就在于可以保存资源。尽管创建新线程、打开文件和其他操作比起来并不是非常昂贵的操作，但是在高并发条件下，开销就会上升得很快。缓存线程和表可以提高效率。

注 2：“打开表”的概念有一点点让人困惑。当不同的查询同时访问某个表，或者某个查询在子查询或者自联接中引用了一个表多次，MySQL 就会认为表被打开了很多次。MyISAM 的索引文件包含了一个计数器，当表打开的时候它的值就会增加，关闭的时候就会减少。这使 MyISAM 知道什么时候表没有被干净地关闭掉。如果它第一次打开一个表，但是计数器的值不是 0，那么就说明这个表没有被干净地关闭。

6.2.6 InnoDB 数据字典

The InnoDB Data Dictionary

InnoDB 自己有对每个表的缓存,叫做“表定义缓存”或者“数据字典”,它是不可配置的。当 InnoDB 打开一个表的时候,它就向字典中添加一个相应的对象。每个表可以使用 4KB 或更多的内存(MySQL 5.1 需要的空间要少得多)。当表关闭的时候,它不会被从字典中删除。

主要的性能问题,除了内存需求之外,就是为表打开和计算统计数据。这个操作是很昂贵的,因为它需要大量的 I/O 操作。和 MyISAM 相比,InnoDB 不会一直在表中保存统计数据。它在每次启动的时候都会重新计算。这个操作在当前版本的 MySQL 中使用了全局互斥量,所以它不能并行地操作。如果有许多表的话,服务器也许需要几个小时来热身,在这期间,它也许除了等待 I/O 操作之外,不能做很多其他的事情。尽管没办法改变这一点,但是还是要知道这一点。

281 这通常只会在有大量(上千或上万)大型表的时候才有问题,它导致 I/O 密集型进程。

如果使用 InnoDB 的 `innodb_file_per_table` 选项(参阅 291 页“配置表空间”),那么对 InnoDB 任何时候能打开的 .ibd 文件的数量还有另外一个限制。这是由 InnoDB 存储引擎处理的,而不是 MySQL 服务器,并且它受 `innodb_open_files` 的控制。InnoDB 打开文件的方式和 MyISAM 不一样。MyISAM 使用表缓存来保存打开表的文件描述符。InnoDB 为每个 .idbw 文件使用了全局文件描述符。如果可以的话,最好把 `innodb_open_files` 设置得足够大,这样服务器就可以保留所有同时打开的 .idb 文件。

6.3 MySQL I/O 调优

Tuning MySQL's I/O Behavior

一些配置选项可以影响 MySQL 把数据同步到硬盘和进行恢复的方式。它们通常对性能有很大的影响,因为这其中牵涉了昂贵的 I/O 操作。它们也代表了性能和数据安全性的折中。通常说来,保证数据被立即而连续地写入磁盘的代价是很高的。如果愿意承担数据不能被真正地写入的风险,就可以增加并发并且减少 I/O 等待的时间,但是你需要决定自己到底能承受多大风险。

6.3.1 MyISAM I/O 调优

MyISAM I/O Tuning

让我们从 MyISAM 如何处理索引开始。MyISAM 通常在每次写入之后就会把索引的改变刷写到磁盘上。如果打算对一个表做很多改变,那么把它们组成一个批处理就会快很多。

做到这点的一种办法是使用 `LOCK TABLES`, 它可以把写入延迟到对表解锁。这对提高性能是很有价值的技巧,因为它让你精确地控制可以延迟写入的查询及写入的时间。可以在语句中精确地定义要延迟的语句。

还可以使用 `delay_key_write` 变量来延迟索引的写入。如果使用了它,修改过的键缓冲区块只有在表关闭的时候才会被写入磁盘(注 3)。它有下面这些可能的选项:

注 3: 表可以因为各种原因被关闭。比如,服务器可能会因为表缓存中没有足够的空间,或者有人执行了 `FLUSH TABLES` 命令,而关闭表。

OFF

MyISAM 在每次写入后就把键缓冲区中修改过的数据块刷写到磁盘上，除非表被 LOCK TABLES 锁住了。

ON

延迟键写入被开启，但是只针对使用 DELAY_KEY_WRITE 选项创建的表。

ALL

所有的 MyISAM 表都使用延迟键写入。

延迟键写入在某些情况下是有帮助的，但是它通常不能带来性能的飞跃。在数据量较小、读取命中率较好并且写入命中率较差的时候，它的用处最大。它也有一些缺点：

- 如果服务器崩溃并且数据块没有被刷写到磁盘上，索引就会损坏。
- 如果许多写入被延迟了，MySQL 就会需要更多的时间来关闭表，因为它要等待缓冲区被刷写到磁盘上。这在 MySQL 5.0 中会导致表缓存被长时间地锁住。
- FLUSH TABLES 可能会需要很长的时间，原因同上。这会导致 LVM 快照或其他备份操作运行 FLUSH TABLES WITH READ LOCK 需要更长的时间。
- 键缓冲区中未被刷写的数据块可能不会给将从磁盘上读取的新块留出空间。因此，查询可能会因为等待键缓冲区释放空间而停止。

除了对 MyISAM 索引 I/O 进行调优，还可以配置 MyISAM 如何从损坏中恢复。myisam_recover 选项控制了 MyISAM 查找和修复错误的方式。可以在配置文件或命令行中设置这个参数。可以使用 SQL 语句参看这个选项的值，但不能对它进行修改（这儿没有写错。这个系统变量和相应的命令行参数的名字不一样）：

```
mysql> SHOW VARIABLES LIKE 'myisam_recover_options';
```

开启这个选项告诉 MySQL 在打开 MyISAM 表的时候检查它的错误，并且如果发现了错误，就尝试修复它。可以给它设置下面的值：

DEFAULT（或者不设置）

MySQL 会尝试修复所有被标记为崩溃及没有被标记为干净地关闭的表。默认设置除了恢复之外，不会做任何事情。和其他大部分变量不一样，DEFAULT 并不会把它设置为编译时给定的值，它只是表示“没有设置”。

BACKUP

让 MySQL 把数据文件备份到一个.BAK 文件中，可以方便随后进行检查。

FORCE

即使.MYD 丢失的数据多于一行，恢复也会继续。

QUICK

除非有被删除的数据块才跳过恢复。这些被删除的行仍然占据空间，并且可以被以后的 INSERT 语句复用。这是有用的，因为对大表进行 MyISAM 恢复通常需要很长的时间。

282

283

可以使用多重设置，中间用逗号隔开。比如，BACKUP、FORCE 将强制恢复并且创建备份。

我们推荐开启这些选项，特别是在仅有一些小的 MyISAM 表的时候。使用被损坏了的 MyISAM 表运行服务器是很危险的，因为它们有时候会损坏更多的数据，甚至让服务器崩溃。然后，如果有大表的话，自动恢复是不切实际的。当它们被打开的时候，就会导致服务器检查和修复所有的 MyISAM 表，这样效率太低。在修复的这段时间内，MySQL 会阻止所有的连接做任何事情。如果有许多 MyISAM 表，那么在启动之后使用 CHECK TABLES 或 REPAIR TABLES 可能会更好一些。这两种方式对于检查和修复表都是很重要的。

启动到数据文件的内存映射访问是另外一种有用的调优手段。内存映射是 MyISAM 能够通过操作系统的页面缓存直接访问到 .MYD 文件，避免了代价较高的系统调用。在 MySQL 5.1 及以上版本中，可以使用 `myisam_use_mmap` 选项打开内存映射。较老版本的 MySQL 只对压缩的 MyISAM 表使用了内存映射。

6.3.2 InnoDB I/O 调优

InnoDB I/O 调优

InnoDB 比 MyISAM 复杂得多。与之相对的是，不仅可以控制它如何恢复，还可以控制它如何打开表及刷写数据，它们极大地影响了恢复和总体性能。InnoDB 的恢复过程是自动的并且在 InnoDB 启动的时候总会运行，但还是可以影响它的行为。更多内容请参阅第 11 章。

先不考虑恢复，假设没有发生任何崩溃或错误，InnoDB 还是有很多值得配置的东西。它有复杂的链式缓冲区和文件，使它可以改进性能并且保证 ACID 属性，并且链上的每一个环节都是可配置的。图 6-1 显示了这些文件和缓冲区。

对于普通使用，一些很重要的配置是 InnoDB 日志文件大小、InnoDB 如何刷写日志缓冲区，以及 InnoDB 如何执行 I/O。

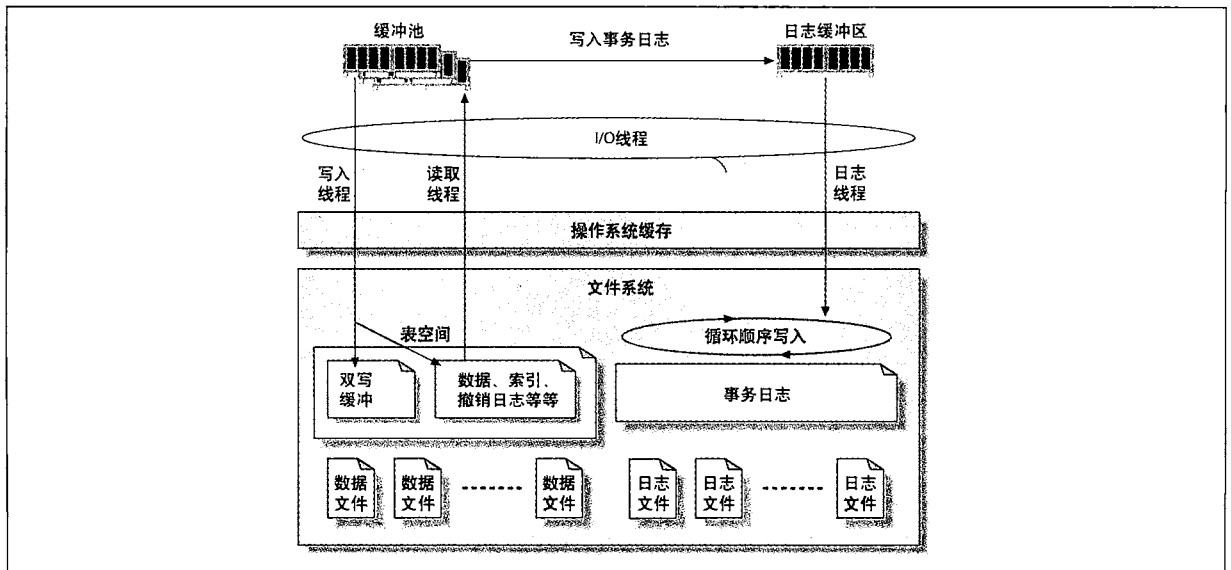


图 6-1: InnoDB 的缓冲区和文件

InnoDB 事务日志

InnoDB 使用日志来减少提交事务的开销。它不是在每次事务提交的时候就把缓冲池刷写到磁盘上，而是记录了事务。事务对数据和索引做出的改变通常会被映射到表空间的随机位置，所以将这些改变写到磁盘上就会引起随机 I/O。作为一条原则，随机 I/O 比顺序 I/O 开销要高得多，因为它需要时间在磁盘上寻找正确的位置，并且还要等磁头移到相应的位置上。

InnoDB 使用自身的日志把随机 I/O 转换为顺序 I/O。一旦日志被记录到磁盘上，事务就是持久的了，尽管这时候改变还没有被写到数据文件中。如果一些坏事发生了（比如断电），InnoDB 可以回放日志并恢复提交了的事务。

当然，InnoDB 最终要把改变写到数据文件中，因为日志的大小是固定的。它以循环的方式写日志，当记录到达日志的底部，就会又从顶部开始。它不会覆盖改变没有被应用到数据文件的记录，因为这会消除提交的事务唯一持久性的记录。

InnoDB 使用后台线程智能地把改变写入到文件中。该线程可以把写入集中在一起，然后以效率更高的顺序写入的方式执行。实际上，事务日志把随机数据文件 I/O 转换为顺序日志文件和数据文件 I/O。把刷写工作变成后台进行可以让查询完成得更迅速，并且在查询负载很大的时候可以对 I/O 系统进行缓冲。

日志文件总体大小由 `innodb_log_file_size` 和 `innodb_log_files_in_group` 控制，并且它们对写入的性能影响极大。这两个文件默认大小都是 5MB，总计为 10MB。对于高性能的负载，这个大小是不够的。日志文件总大小的上限是 4GB，但是即使是写入负载极高的查询也只需要几百 MB（比如总共 256MB）。接下来的章节解释了如何为负载找到合适的大小。

InnoDB 用多个文件组成一个循环日志系统。通常不用改变默认的日志数量，只须改变每个日志文件的大小就可以了。为了改变日志文件的大小，先关闭 MySQL，然后移走旧日志，再重新配置大小，最后重新启动 MySQL 就行了。要确保干净地关闭 MySQL，否则日志文件中就有可能还保留着需要被应用到数据文件的记录。当重新启动服务器的时候要注意查看错误日志。在重新启动之后，就可以删除掉老的日志。

日志文件大小和日志缓冲。为了决定日志文件的理想大小，需要衡量通常数据改变的开销和崩溃时恢复的时间。如果日志太小，InnoDB 将会设置更多的检查点，并且导致更多日志写入。在极端情况下，写入查询有可能会停下来，等待日志上的记录被应用到数据文件上。另一方面，如果日志太大，InnoDB 在恢复的时候可能会做大量的工作。它会极大地增加恢复的时间。

数据大小和访问模式也会影响恢复时间。假设有 1TB（译注 1）数据并且有 16GB 缓冲池，总日志是 128MB。如果在缓冲池中有许多不干净的页面（比如，更改没有被应用到数据文件的页面），并且它们在整个 1TB 数据中均匀分布，那么从崩溃中恢复就会花很长时间。InnoDB 将会不得不扫描日志、检查数据文件，并且按照需要把改动应用到文件上。这意味着大量的读写！在另一方面，如果更改是局部化的，比如说只有几 GB 数据经常被改动，恢复就会很快，即使数据和日志文件很大也是这样。恢复时间也依赖于典型更改的大小，它和数据行的平均长度有关。较短的行可以在日志中存储更多的改动，所以 InnoDB 在恢复的时候就会回放更多的改动。

在 InnoDB 改变数据的时候，它会把这次改动的记录写到日志缓冲里面。日志缓冲被保存在内存中。缓冲写满、事务提交或每一秒钟，不管那种情况先发生，InnoDB 都会把缓冲区写到磁盘上的日志文件中。如果有大型事务，就可以增加缓存文件（默认是 1MB）来减少 I/O 动作。控制缓冲大小的变量叫 `innodb_log_buffer_size`。

译注 1：1024GB。

不需要把缓冲区变得很大。推荐值是 1MB 到 8MB。除非要写入大量的巨型 BLOB 记录, 否则这个大小就足够了。日志相对 InnoDB 的正常数据要紧凑得多。它们不是基于页面的, 所以它们不会在存储数据的时候浪费整个页面。InnoDB 也会使日志记录尽可能短。它们有时甚至会被像 C 函数的函数名和参数那样存储。

可以通过检查 SHOW INNODB STATUS 命令的 LOG 部分检测 InnoDB 的日志和日志缓冲 I/O 性能, 还可以通过观察 InnoDB_os_log_written 的值了解 InnoDB 向日志文件写入了多少数据。一个好的法则就是观察 10 秒到 100 秒时间间隔内的数据, 并且注意最大值。可以使用这个值来判断日志缓冲大小是否合适。例如, 如果最大数据是每秒写入 100KB, 那么 1MB 的日志缓存可能就足够了。

也可以使用这个指标来决定日志文件的合适大小。如果最大值是每秒 100KB, 256MB 日志文件就可以存储至少 2560 秒的日志记录, 这很可能足够了。参阅第 565 页的“显示 INNODB 状态”了解如何监控和解释日志及缓冲区的状态。

InnoDB 如何刷写日志缓冲。当 InnoDB 把日志缓冲刷写到磁盘上的日志文件时, 它会用一个互斥量锁定缓存, 把缓存写到应有的位置, 然后把剩下的记录移到缓存的前端。一种可能性就是当互斥量被释放时, 不止一个事务打算刷写自己的日志。InnoDB 有群体提交特性, 它可以利用一次 I/O 操作把所有的请求提交到日志中。但是对于 MySQL 5.0, 如果二进制日志被激活, 这个功能就失效了。

日志缓冲区必须被刷写到持久性存储中, 以保证提交了的事务能完全持久化。如果比起持久性, 更在意性能, 就可以改变 innodb_log_at_trx_commit 的值来控制日志缓存被刷写到什么地方及刷写的频率。可能的设置如下:

- 0 把日志缓存写到日志文件中, 并且每秒钟刷写一次, 但是在事务提交的时候不进行任何动作。
- 1 将日志缓冲写到日志文件中, 并且在事务提交的时候把缓存刷写到持久性存储中。这是默认(最安全)的设置。它保证不会遗失任何提交的事务, 除非磁盘或操作系统“假冒”了刷写操作。
- 2 在每次提交的时候把日志缓冲写到日志文件中, 但是不进行清理。InnoDB 安排每秒钟清理一次。它和 0 最大的区别(这也使得 2 更好)是 2 在 MySQL 进程崩溃的时候不会丢失任何事务。但是如果日志服务器崩溃或失去电力, 还是有可能丢失事务。

重要的是知道把日志缓冲写入日志文件和把日志刷写到持久性存储中的区别。在大多数操作系统中, 把缓冲写入日志只是简单地数据从 InnoDB 的内存缓冲区移到操作系统的缓存中, 它也在内存中。它实际上不会把数据写入持久性存储中。因此, 设置 0 和 2 通常导致在崩溃或电力故障的时候最多失去一秒钟的数据, 因为这时数据可能只在操作系统的缓存中。之所以说“通常”, 原因是 InnoDB 会每秒钟把日志文件写到磁盘上一次, 但是在某些情况下也有可能失去多于 1 秒的数据, 例如刷写被停住了。

相反的是, 将日志刷写到持久性存储中意味着 InnoDB 要求操作系统把数据刷到缓存外部并且确保写入到磁盘上。这会阻止 I/O 调用, 直到数据被完全写入。由于将数据写到磁盘比较慢, 所以当 innodb_flush_log_at_trx_commit 被设置到 1 时, 它会显著地降低 InnoDB 每秒可以提交的事务。现在的高速驱动器(注 4)每秒只能执行几百次真正的磁盘事务, 这是由驱动器旋转和寻址时间所决定的。

有时硬盘控制器或操作系统会假冒清写动作, 它会把数据放入另外一个缓存中, 比如磁盘自己的缓存。这样做速度较快但是很危险, 因为数据在驱动器失去电力的时候还是会丢失。这甚至比把 innodb_flush_log_at_trx_commit 设置为 1 之外的值更糟糕, 因为它不仅仅是导致丢失事务, 还能导致数据损坏。

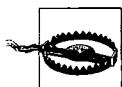
注 4: 我们谈论的是普通的磁碟式硬盘, 而不是固态硬盘。固态硬盘的性能特点完全不一样。

把 `innodb_flush_log_at_trx_commit` 设置为 1 之外的值能导致丢失事务。但是,如果不在意持久性(Durability ACID 中的 D),把它设置为其他值也是有用的。也许你只需要 InnoDB 的其他特性,比如聚集索引、防止数据损坏及行级锁。完全为了性能原因使用 InnoDB 代替 MyISAM 也是常见的。

高性能事务的最佳配置是把 `flush_log_at_trx_commit` 设置为 1,并且将日志文件放在有备用电池的写入缓存的 RAID 上。这不仅安全,速度也很快。更多 RAID 的内容请参阅第 317 页的“RAID 性能优化”。

InnoDB 如何打开并清写日志和数据文件

`innodb_flush_method` 选项让你可以配置 InnoDB 实际与文件系统进行交互的方式。除了写数据之外,它还可以影响 InnoDB 如何读取数据。Windows 和非 Windows 上这个选项的值是互斥的。在 Windows 系统上,只能使用 `async_unbuffered`、`unbuffered` 和 `normal`,不能使用其他任何值。在 Windows 系统上的默认值是 `unbuffered`,其他系统是 `fdatsync`。(如果 `SHOW GLOBAL VARIABLES` 显示该变量的值为空,那就意味着它被设置成了默认值。)



警告: 改变 InnoDB 如何执行 I/O 操作会极大地影响性能,要仔细地评测它们。

下面是一些可能的值:

`fdatsync`

它是非 Windows 系统上的默认值。InnoDB 使用 `fsync()` 函数来刷写数据和日志文件。

InnoDB 通常使用 `fsync()` 来代替 `fdatsync()`,尽管它们的意思好像是相反的。`fdatsync()` 和 `fsync()` 相似,除了它只刷写文件的数据,而不包括元数据(最后修改时间等)。因此,`fsync()` 能导致更多的 I/O 操作。但是 InnoDB 的开发人员非常保守,并且他们发现 `fdatsync()` 在某些情况下会导致崩溃。InnoDB 决定哪种方法能被安全地使用,一些选项在编译时就被设置好了,有些在运行时进行设置。它会使用最快的方法。

使用 `fsync()` 的缺点就是操作系统至少会缓存一些数据在自己的缓存中。在理论上,这是有点浪费的双缓冲,因为 InnoDB 能比操作系统更智能地管理自己的缓冲区。但是,最终的影响和系统及文件系统非常相关。如果双缓冲可以让文件系统更智能地做 I/O 计划和进行批处理,那它也不是什么坏事。一些文件系统和操作系统能累计写入并一起执行它们,还能按照效率进行排序,或者并行地把它们写到多个设备中。它们也会做读取前的优化,比如指导磁盘预读下一个顺序块。

有时这些优化是有帮助的,但是有时又不是。如果好奇自己机器上的 `fsync()` 到底是什么版本,可以使用 `fsync(2)` 读取系统的指南页 (Manpage)。

`innodb_file_per_table` 导致每个文件都被单独地使用了 `fsync()` 函数,这意味着想多个表写入不能被合并到单个 I/O 操作中。这可能需要 InnoDB 执行较多的 `fsync()` 操作。

`O_DIRECT`

InnoDB 根据系统和数据文件使用 `O_DIRECT`,或者 `directio()` 标志。该选项不会影响日志文件并且不是在所有 Unix 系统上都可用。至少 Linux、FreeBSD 和 Solaris (5.0 及以上版本) 支持它。和 `O_DSYNC` 标志不同,它会影响读写。这个设置还是使用 `fsync()` 把文件刷写到磁盘,但是它告诉操作系统不要缓存数据,也不要使用提前读取。它完全禁止了操作系统的缓存并且使所有的读写动作直接到存储设备,避免了

双缓冲。

在大多数系统上，它通过调用 `fcntl()` 在文件描述符中设置 `O_DIRECT` 标志。所以可以通过 `fcntl(2)` 读取指南页了解详细情况。在 Solaris 上，该选项使用了 `directio()`。

该设置不能禁止 RAID 卡的提前读取功能。它只能禁止操作系统或文件系统的提前读取功能。

在使用 `O_DIRECT` 的时候通常不应该禁止 RAID 卡的写入缓存，因为这是保持良好性能的唯一途径。当在 InnoDB 和实际存储设备之间没有缓存的时候使用 `O_DIRECT` 会极大地降低性能。

这个设置能导致服务器热身时间增加，特别是操作系统缓存非常巨大的时候。它也能使小缓存池（例如，默认大小的缓存池）比缓冲过的 I/O 慢很多。这是因为操作系统通过把数据保存在自己的缓存中以“帮助”数据移出。如果需要的数据不在缓存池中，InnoDB 将会直接从磁盘上读取它。

该设置不会对使用 `innodb_file_per_table` 有更多的危害。

`O_DSYNC`

该选项在对日志文件调用 `open()` 使用了 `O_SYNC` 标志。这使所有的写入都是同步的，换句话说，写入动作只有在数据被写入磁盘的时候才会返回。该选项不会影响数据文件。

在 `O_SYNC` 和 `O_DIRECT` 标志之间的区别是 `O_SYNC` 不会在操作系统层面禁止缓存。因此，它不会避免双缓冲，并且它不会直接写入磁盘。使用 `O_SYNC`，写入修改了缓存中的数据，然后把它发送到磁盘上。

尽管使用 `O_SYNC` 同步写入听上去很像 `fsync()`，但是它们在操作系统和硬件层面的实现是很不一样的。当使用 `O_SYNC` 的时候，操作系统会把“使用同步 I/O”的标志传递到硬件层，告诉设备不要使用缓存。在另一方面，`fsync()` 告诉操作系统把修改过的缓存写到设备上，然后告诉设备如果可能的话就清理掉自己的缓存。所以它确保数据已经被记录到了物理介质上。另外一个区别就是使用 `O_SYNC`，每个 `write()` 或者 `pwrite()` 操作在完成之前，都会阻塞调用过程。相反地，不使用 `O_SYNC` 标志调用 `fsync()` 会在缓存中累计写入（这使每个写入更快），然后再一次性地进行刷写。

还有，该选项设置了 `O_SYNC` 标志，而不是 `O_DSYNC` 标志，因为 InnoDB 开发人员发现了使用 `O_DSYNC` 的问题。`O_SYNC` 和 `O_DSYNC` 类似于 `fsync()` 和 `fdatasync()`：`O_SYNC` 同步数据和元数据，但是 `O_DSYNC` 只同步数据。

`Async_unbuffered`

这是 Windows 系统的默认值。该选项让 InnoDB 对大部分写入都使用无缓冲的 I/O。例外情况就是它在 `innodb_flush_log_at_trx_commit` 被设置到 2 的时候对日志文件使用了有缓冲的 I/O。

该选项使 InnoDB 在 Windows 2000、Windows XP 及以上版本中对读写都使用操作系统原生的异步（层叠）I/O。在较老版本的 Windows 上，InnoDB 使用自己的异步 I/O，它使用了线程。

`Unbuffered`

只适用于 Windows。该选项和 `async_unbuffered` 类似，但是不使用原生异步 I/O。

`Normal`

只适用于 Windows。该选项让 InnoDB 不使用原生异步 I/O 或者无缓冲 I/O。

仅能用于开发。这两个选项没有文档支持，对产品服务也不安全，不应该使用它们。

如果你的 RAID 控制器的写入缓存有备用电池，我们推荐使用 `O_DIRECT`。如果不是，默认值或 `O_DIRECT` 都有可能是最佳选择，这依赖于应用程序。

可以配置 Windows 中 I/O 线程的数量，但是在其他平台上都不行。把 `innodb_file_io_threads` 设置为大于 4 将会导致 InnoDB 为数据 I/O 创建更多的读写线程。一般只有 1 个插入缓冲线程和 1 个日志线程。所以如果它的值是 8 的话，那就有 1 个插入缓冲线程、1 个日志线程、3 个读取线程和 3 个写入线程。

InnoDB 表空间

InnoDB 把数据保存在表空间中。表空间实际上是跨越了磁盘上的一个或多个文件的虚拟文件系统。InnoDB 出于很多考虑使用了表空间，而不仅仅是为了存储表和索引。它保留了自己的撤销日志（老的数据行的版本）、插入缓存、双写缓存（下一节将会介绍），以及表空间的其他内部结构。

291

配置表空间。可以使用 `innodb_data_file_path` 定义表空间文件。这些文件都在 `innodb_data_home_dir` 定义的目录中，下面是一个例子：

```
innodb_data_home_dir = /var/lib/mysql/
innodb_data_file_path = ibdata1:1G;ibdata2:1G;ibdata3:1G
```

上面的例子在 3 个文件中创建了 3GB 的表空间。有时人们在想是否可以跨越多个驱动器使用文件来分摊负载，就像这样：

```
innodb_data_file_path = /disk1/ibdata1:1G;/disk2/ibdata2:1G;...
```

这确实把文件放在了不同的目录中，本例中是不同的驱动器。InnoDB 实际上只是把这些文件连接了起来。因此，用这种办法不会有什么好处。InnoDB 会先把第 1 个文件写满，然后是第 2 个，依次类推。负载其实并没有被摊开。RAID 控制器能更智能地分摊负载。

为了使表空间在空间耗尽之后可以自动增长，可以用下面的命令让最后一个文件自动延伸：

```
...ibdata3:1G:autoextend
```

默认行为是创建 10MB 的单个延伸文件。如果让文件可以自动延伸，一种不错的方式就是给表空间的值设定一个上限，使之不会增长得过大，因为它一旦长大了，就不会自动减小。例如，下面的例子限制延伸文件最大为 2GB：

```
...ibdata3:1G:autoextend:max:2G
```

管理单个表空间也许是一件费力气的事情，尤其是它能自动增长，而你又想回收空间（出于这个原因，我们建议你禁止自动延伸）。回收空间的唯一办法就是将数据进行转储、关闭 MySQL、删除所有的文件、改变配置、重新启动、让 InnoDB 创建新文件并且恢复数据。InnoDB 对表空间的控制是很严的，不能简单地移除文件或改变大小。如果表空间崩溃了的话，InnoDB 就不会启动。这和它对日志文件的严格限制一样。如果你习惯了在 MyISAM 中随便移动数据，在这儿可要当心些。

在 MySQL 4.1 及以上版本中，`InnoDB_file_per_table` 选项使 InnoDB 为每一个表使用一个文件。它在数据库目录中以“表名.ibd”保存数据。这使得删除表后回收数据变得比较容易，并且它对于把表分布到多个磁盘上也

很有用处。但是，将数据放在多个文件中能导致浪费更多的存储空间，因为它把单个 InnoDB 表空间的碎片都放在了 .ibd 文件中。

292 这对于很小的表尤其会成为一个问题，因为 InnoDB 的页面大小是 16KB。即使表只有 1KB 数据，它也会需要至少 16KB 的磁盘空间。

即使开启了 `innodb_file_per_table` 选项，还是需要为撤销日志和其他系统数据定义主表空间。（如果没有在里面保存所有的数据，它会比较小。但是关闭自动延伸还是一个好主意，因为只有重新加载数据才能减少文件大小。）同样，不能简单地通过拷贝文件来移动、备份或恢复表。也有可能做到这一点，但是需要额外的步骤，并且肯定不能在服务器之间拷贝数据。更多详情请参阅第 500 页的“恢复原始文件”。

一些人喜欢使用 `innodb_file_per_table`，仅仅是因为它的可管理性和可见性。例如，通过检查文件得到表的大小比通过 `SHOW TABLE STATUS` 要快得多，后者需要锁定表并且扫描缓冲池，以了解有多少页面被分配给了表。

我们也应该注意到实际上不用把 InnoDB 文件保存在传统的文件系统中。和很多传统的数据库服务器一样，InnoDB 提供了使用原始设备，比如未格式化的分区来存储数据的能力。然而，现在的文件系统能高效地处理大型文件，所以没有必要使用这种能力。使用原始设备可能会把性能提高几个百分点，但是我们不认为这点性能提升可以弥补不能按照文件来操作数据的缺点。当把数据保存在原始分区（Raw Partition）中时，不能对它使用 `mv`、`cp` 或者其他的工具。我们也考虑了快照能力，比如由 GNU/Linux 的逻辑卷管理器（LVM, Logical Volume Manager）提供的功能，有很大的好处。可以把原始设备放到逻辑卷中，但这违反了初衷，因为这已经不是真正的原始设备了。最后要说的是，由使用原始设备带来的一点点性能提升并不划算。

旧数据版本和表空间。InnoDB 的表空间在写入负荷很重的环境中会增长得很大。如果事务长时间处于打开状态（即使它没有做任何工作），并且它们正在使用默认的 `REPEATABLE READ` 事务隔离层，InnoDB 将不能删除老的数据，因为未提交的事务还需要它们。InnoDB 将老的数据保存在表空间中，所以当更多的数据被更新时，它就会继续增长。有时问题不是出在没有提交的事务上，而是出在工作负载上：清理进程是单线程的，它跟不上需要被清理的老数据的数量。

在这两种情况下，`SHOW INNODB STATUS` 的输出有助于锁定问题。查看 `TRANSACTIONS` 的第 1 和第 2 行，它会显示当前事务的数量及指出哪个的清理工作已经完成了。如果它们的区别很大，那么就说明有许多没被清理的事务，下面是个例子：

```
293 -----
TRANSACTIONS
-----
Trx id counter 0 80157601
Purge done for trx's n:o <0 80154573 undo n:o <0 0
```

事务标识符是一个由两个 32 位整数组成的 64 位整数，所以需要去做一点点数学计算来了解它们的区别。在这个例子中，计算非常容易。数字的高位都是 0，所以就有 $80157601 - 80154573 = 3028$ 个潜在的未被清理的事务（Innotop 可以帮你做这个数学计算）。这儿说“潜在的”，是因为大的差别并不一定是指有很多未被清理的行。只有改变了数据的事务才会创建旧数据，有可能很多事务没有改变数据（同样，有可能一个事务就改变了很多数据）。

如果有很多未被清理的事务并且表空间因它而增长，就可以强制 MySQL 变慢，以使清理线程能跟上数据的变化。这听上去不怎么好，但是也没什么替代的办法。否则，InnoDB 就会不停地写入数据并且填充磁盘，直到耗尽磁盘空间或让表空间达到规定的上限。

为了减缓写入，可以把 `innodb_max_purge_lag` 变量设置为 0 之外的值。这个值表示在等待清理的事务的最大数量，一旦超过这个值，InnoDB 就会延迟更新数据的事务，要知道自己的负载才能决定这个值的最佳大小。举个例子，如果事务平均影响 1KB 数据，并且能容忍 100MB 未清理的数据，那么这个值就是 1 000 000。

要记住未被清理的行会影响所有的查询，因为它会让表和索引很快变大。如果清理线程跟不上节奏，性能就会大打折扣。设置 `innodb_max_purge_lag` 也会降低性能，但是它的危害比前者小。

双写缓存

InnoDB 在对页面进行部分写入的时候使用了双缓冲，以防止数据损坏。部分写入发生在磁盘写入没有全部完成，并且只有 16KB 页面的一部分被写入的时候。有许多原因（崩溃、缺陷等）会导致数据被部分写入。双写缓存在这种情况下保护了数据。

双缓冲是表空间中一个特殊的保留区域，大小足够在一个连续块中容纳 100 个页面。它在本质上是最近写入页面的备份。当 InnoDB 把页面从缓冲池中清写到磁盘上时，它会先把它们写入（或者清写）到双缓冲中，然后再写入到真正的地方。这确保每次写入的原子性和可持续性。

这不是意味着每个页面都被写了两次吗？是的，确实是两次，但是因为 InnoDB 顺序地把一些页面写入双缓冲，然后再调用 `fsync()` 把它们同步到磁盘上，所以对性能的影响比较小——通常是几个百分点。更重要的是，这个策略使日志文件高效得多。因为双写缓冲为 InnoDB 保证数据不受破坏提供了强有力的保障，所以 InnoDB 的日志记录就不需要包含完整的页面，而更像是页面的二进制增量数据。

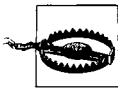
如果有部分页面被写入了双写缓存，那么原始页面还将继续留在磁盘的本来位置上。当 InnoDB 恢复的时候，它会使用原始页面来代替双写缓存中被破坏的页面。但是，如果双写缓存成功并且写到页面的真实位置失败了，InnoDB 就会在恢复的时候使用双写缓存中的拷贝。InnoDB 知道每个被损坏了的页面，因为每个页面的最后都会有一个校验码。校验码最后才会被写入页面。所以如果页面的内容和校验码不匹配，那么就说明页面已经被损坏了。因此，在恢复的时候，InnoDB 只是读取双写缓存中的页面并且验证校验码。如果页面的校验码不正确，它就会从原始位置读取页面。

在某些情况下，双写缓存并不是必需的，例如在从服务器上可以禁用它。同时，一些文件系统（比如 ZFS）自身会做同样的事情，那么 InnoDB 就不用做重复的工作了。可以通过把 `innodb_doublewrite` 设置为 0 禁用双写缓存。

另外的 I/O 调优

`Sync_binlog` 选项控制了 MySQL 如何把二进制日志刷写到磁盘。它的默认值是 0，这意味着 MySQL 不会执行任何刷写操作，并且何时把日志写到持久性存储设备上取决于操作系统。如果该值大于 0，它就规定了在把二进制日志刷写到磁盘期间可以运行多少次写入（如果 `autocommit` 被设置为 1，那么每次写入就是单条语句，否则就是一个事务）。一般很少把它设置为 0 或 1 之外的值。

如果没有把 `sync_binlog` 设置为 1，那么就可能导致二进制日志和事务的数据不同步。这会破坏复制并且使按时间点进行恢复变得不可能。然而，把它设置为 1 带来的安全性需要很高的代价。同步二进制日志和事务日志要求 MySQL 把两个文件刷写到两个不同的磁盘位置上。这会导致相对较慢的磁盘搜索。



警告：如果在 MySQL 5.0 及更高的版本上使用二进制日志和 InnoDB，并且尤其是在从低版本升级到高版本的情况下，那么就要特别小心 XA 事务这一新特性。它被设计成用于在不同存储引擎和二进制日志之间同步事务提交，但是它也禁用了 InnoDB 的群体提交功能。这会极大地降低性能，因为提交事务的时候可能会导致大量的 `fsync()` 调用。可以通过禁用二进制日志和把 `innodb_support_xa=0` 设置为 0 禁用 XA 支持来解决这个问题。如果使用了有备用电池的 RAID 缓存，每个 `fsync()` 调用都会更快，所以这可能不会成为一个问题。

295

和 InnoDB 日志文件一样，把二进制日志文件放入有备用电池的 RAID 卷会极大地提高性能。

有一点和性能无关的提示：如果想使用 `expire_logs_days` 选项自动地删除老的二进制日志，就不要使用 `rm` 来删除它们。否则服务器会弄不清楚到底是哪一个起作用，于是就不会自动地删除它们，并且 `PURGE MASTER LOGS` 也会停止工作。在发现自己有这个问题的时候，解决的办法就是手动地使用磁盘上的文件列表和 `hostname-bin.index` 文件进行重新同步。

在第 7 章会深入讨论 RAID，但是在这儿有一点值得重申，那就是质量优良的 RAID 控制器，在具有备用电池的写入缓存并且使用了回写策略的情况下，每秒可以处理上千次写入，并且还能提供其他存储服务。数据被写入到了有电池的缓存中，所以即使系统失去了电力，数据也不会丢失。当电力供应恢复的时候，RAID 控制器就会在磁盘可用之前把数据写入磁盘。因此，有足够写入缓存的 RAID 控制器能极大地提高性能并且这会很划算的投资。

6.4 MySQL 并发调优

Tuning MySQL Concurrency

当 MySQL 在高并发条件下工作的时候，你可能会遇到以前未曾遇到过的性能瓶颈。接下来的章节解释了如何检测问题，并且如何在 InnoDB 和 MyISAM 遇到高并发的时候得到最佳性能。

6.4.1 MyISAM 并发调优

MyISAM Concurrency Tuning

需要很仔细地控制同时进行的读写，以避免读取到不连续的数据。MyISAM 在某些条件下允许并发插入和读取，并且它让你可以“调度”某些操作，以尽可能少地阻止工作。

在了解 MyISAM 的并发设置之前，重要的是要了解 MyISAM 如何删除和插入行。删除操作不会重新安排整个表，它们只是把行标记为已经删除，并且在表中留下了一些“洞”。MyISAM 在可能的情况下会优先使用这些“洞”，为插入复用空间。如果表是完整的，它就会把新的行拼接在表的最后。

即使 MyISAM 有表级别的锁，它也能在读取的同时把行拼接到表尾。它通过禁止读取最后一行做到了这一点。这避免了不连续的读取。

但是，当表中间的数据改变的时候，要提供连续读取就困难得多。MVCC 是最通用的解决这个问题的办法：它在创建新版本数据的同时提供老版本数据读取。

MyISAM 不支持 MVCC，所以它只有在到达表尾的时候才允许并发插入。

可以使用 `concurrent_insert` 变量配置 MyISAM 的并发插入行为，它下面的值：

- 0 MyISAM 不允许并发插入，每一次插入都会把表锁住。
- 1 默认值。只要表中没有空缺，MyISAM 就允许并发插入。
- 2 该值在 MySQL 5.0 及更高的版本可用。它强制并发插入到表尾，即使表有空缺也不例外。如果没有线程从表中读取数据，MySQL 就会把新数据插入到空缺中。使用了该设置，表的碎片会增多，所以需要更经常地对表进行优化。

可以配置 MySQL 把一些操作延迟，然后合并到一起执行。例如，可以使用 `delay_key_write` 延迟写入索引。这会带来一些明显的矛盾：立即写入索引（安全但是代价很高），或者等待写入并希望写入前不要断电（更快，但是如果断电的话就会导致大规模的索引损坏，因为索引文件已经明显过期了）。也可以使用 `low_priority_updates` 让 INSERT、REPLACE、DELETE 及 UPDATE 比 SELECT 的优先级更低。这等同于全局地给 UPDATE 使用 `LOW_PRIORITY` 修饰符。更多内容请参阅第 195 页的“查询优化器提示”。

最后，尽管 InnoDB 的扩展性问题被提及的很多，但 MyISAM 实际在很长时间内都有互斥量的问题。在 MySQL 4.0 和更早版本中，有全局互斥量保护键缓冲的任何 I/O 行为，它在多 CPU 和多磁盘的时候带来了扩展性问题。MySQL 4.1 的键缓冲代码已经被改进过，并且不再有问题，但是它还是为每一个键缓冲保持了一个互斥量。当一个线程从键缓冲把键数据块拷贝到本地存储中，而不是从磁盘读取时，它会成为一个问题。磁盘的瓶颈已经没有了，但是在访问键缓冲中的数据时瓶颈依然存在。有时可以使用多个键缓冲绕过这个问题，但是这种方法并不总是奏效。例如，在只包含了一个索引的时候就没办法解决这个问题。这样的话，多 CPU 机器上的并发 SELECT 查询会比单 CPU 的机器慢很多，即使只有这一个查询在运行也是如此。

6.4.2 InnoDB 并发调优

InnoDB Concurrency Tuning

InnoDB 是为高并发设计的，但它并不完美。InnoDB 的结构仍然基于有限内存、单 CPU 和单磁盘系统。InnoDB 某些方面的性能在高并发条件下下降得很快，并且唯一的解决办法就是限制并发。可以经常检查 `SHOW INNODB STATUS` 输出中的 `SEMAPHORES` 部分来确认是否发生了并发问题。更多内容请参阅第 566 页的“SEMAPHORES”。

297

InnoDB 用自己的“线程调度”程序来控制线程如何进入 InnoDB 的内核访问数据，以及一旦进入内核之后可以执行的动作。控制并发最基本的方式是使用 `innodb_thread_concurrency` 变量，它限制了一次有多少线程能进入内核。0 表示不限制进入内核的数量。如果有 InnoDB 并发问题，该变量是最重要的配置项。

没有办法为所有的架构和工作负载确定最佳的并发数量，在理论上，可以用下面的公式进行计算：

$$\text{并发} = \text{CPU 的数量} \times \text{磁盘的数量} \times 2$$

但是在实际中，使用更小的值往往会更好。为了知道适合自己系统的最佳值，需要进行试验和测试。

如果内核中已经有了允许数量的线程，那么线程就不能再进入内核了。InnoDB 采用了一种两阶段的过程来保证线程可以尽可能高效地进入内核。这种策略减少了操作系统引起的上下文切换带来的开销。线程首先睡眠 `innodb_thread_sleep_delay` 所规定的微秒数，然后再次进行尝试。如果还是不能进入，它就会进入一个等待线程的队列中并且把控制权交给操作系统。

第一阶段默认的睡眠时间是 10 000 微秒。当有很多线程都处于“正在等待进入队列”这一状态时，改变这个值有助于高并发性系统。默认值在有大量小查询的时候会太大了，因为它给查询增加了 10 毫秒延时。

一旦线程进入了内核，它就会得到一个确定的数字作为“凭据”，它再次进入内核的时候，就不会再进行任何的并发检查。该数字限定了它再次回到等待队列之前能做多少工作。InnoDB_concurrency_tickets 选项控制了凭据的数量。除非有大量的运行极长时间的查询，否则我们极少改动这个选项。凭据只是为每个查询授权，而不是为每个事务授权。一旦查询结束，凭据就会被丢掉。

除了缓冲池和其他结构的瓶颈，在提交阶段还有另外一种形式的并发瓶颈，也就是刷写操作造成的密集 I/O 操作。InnoDB_commit_concurrency 变量决定了某一时刻有多少线程能进行提交。当 innodb_thread_concurrency 被设置到了一个较低的值，造成大量线程状况不佳时，设置该选项会有帮助。

InnoDB 团队正在解决这些问题，在 MySQL 5.0.30 和 MySQL 5.0.32 中有了较大的改进。

6.5 基于工作负载调优

Workload-based Tuning

对服务器调优的最终目的是按照特定的工作负载进行定制。这不仅仅需要了解查询，还需要深刻了解服务器各种活动的数据、类型及频率，以及其他的一些活动，比如到服务器的连接和刷写数据表。还需要了解如何监控和解释 MySQL 以及操作系统的行为和状态。更多内容可以参阅第 7 章和第 14 章。

第一件要做的事情就是熟悉服务器。了解运行在它上面的查询。使用 innotop 或其他工具对它进行监控。这不仅仅有助于知道服务器的总体情况，也有助于知道查询正在做什么。获取这种知识的一种方式就是使用一个脚本（Innotop 内建了该功能）聚合 SHOW PROCESSLIST 输出中 COMMAND 栏的内容，或者就用肉眼进行观测。要找到在某种特殊情况下花费了大量时间的线程。

在服务器满负荷运行的时候试着查看一下进程列表是了解何种类型的查询遇到了最大困难的最佳时机。例如，有许多查询把结果拷贝到了临时表吗？或者正在对结果进行排序？如果是的话，那么就应该检查临时表和排序缓冲的配置（也许也要优化查询）。

我们常常推荐我们为 MySQL 日志开发的补丁，它能显示查询所做事情的大量信息，并且还能让你更详细地分析负载。这些补丁被包含在 MySQL 官方发布中，所以它们可能已经在你的机器中了。具体内容请参阅第 65 页的“更好地控制日志”。

6.5.1 优化 BLOB 和 TEXT 负载

Optimizing BLOB and TEXT Workloads

BLOB 和 TEXT 对于 MySQL 来说是特殊的负载（为了简单起见，我们把 BLOB 和 TEXT 统称为 BLOB，因为它们属于同一种数据类型）。BLOB 有一些局限，所以服务器不得不特殊地处理它们。一个最重要的考虑就是服务器不能为 BLOB 使用内存中的临时表。因此，如果某个查询需要为 BLOB 值使用数据表，不管它是否很小，都会在磁盘上进行。这效率很低，尤其是对小而快的查询而言，临时表可能是查询开销中最大的部分。

有两种办法可以解决这个问题：（1）利用 SUBSTRING 函数把值转换为 VARCHAR（具体内容请参阅第 84 页的“字符串类型”）；（2）让临时表运行得更快。

让临时表更快的最佳方式就是把它们放在基于内存的文件系统中（GNU/Linux 中的 `tmpfs`）。

这减少了一些开销，但仍然比使用内存中的表慢很多。使用基于内存的文件系统有一些帮助，因为操作系统会避免向磁盘写入数据（注 5）。普通的文件系统也被缓存到了内存中，但是操作系统会几秒钟就向磁盘写一次数据。`tmpfs` 文件系统永远都不会刷写数据。它也是为较低开销和简单性而设计的。例如，这种文件系统不需要为恢复做准备，这使它运行得更快。

控制临时表位置的服务器设置是 `tmpdir`。要监控文件系统被写满的程度，以保证临时表有足够的空间。如果需要，可以定义几个临时表位置，MySQL 会循环使用它们。

如果 BLOB 列非常大，并且使用的是 InnoDB，那么就应该增加 InnoDB 的缓冲区大小。本章前面的节有更多这方面的内容。

对于很长的长度可变的列（比如，BLOB、TEXT 和很长的字符列），InnoDB 会在页面中存储一个 768 字节的前缀（注 6）。如果列的值大于前缀的长度，InnoDB 可能会分配外部的存储空间来保存其余的数据。InnoDB 会为数据分配 16KB 大小的整个页面，并且每一个列都会有自己的页面（列不会共享页面）。InnoDB 一次为一个列分配一个页面，直到页面达到 32 个，然后它就会一次性分配 64 个页面。

注意到我们只是说 InnoDB 可能会分配外部存储空间。如果行的总长度，包括长列的完整值，小于 InnoDB 的最大行长度（稍小于 8KB），那么即使长列的值超过了前缀长度，InnoDB 也不会分配外部存储空间。

最后，当 InnoDB 更新外部存储空间中的长列时，它不会对它进行原地更新。相反地，它会把值放到外部存储空间中的新地方，然后把老的数据删除。

这些行为会造成下面的后果：

- 长列可能会浪费很多 InnoDB 空间。例如，如果列的值只比行的最大长度多一个字节，InnoDB 也会用整个页面来存储其余的数据，这样就浪费了页面的大部分空间。同样地，如果值大小稍大于 32 个页面，它在磁盘上就实际会用 96 个页面。
- 外部存储禁用了适应性哈希索引，它需要比较列的完全长度来验证数据正确性。（哈希让 InnoDB 能很快地找到自己“猜测”的结果，但是它必须检测自己的“猜测”是否正确。）因为适应性哈希索引全部在内存中并且它直接从缓存池中最经常被访问的数据上构造出来，它不能使用外部存储。
- 长值会让无法使用索引的有 WHERE 子句的查询运行缓慢。MySQL 在应用 WHERE 子句之前会读取所有的列，所以它可能会让 InnoDB 读取大量的外部存储，然后检查 WHERE 子句并把读取的数据丢掉。选择不需要的列永远都不是一个好主意，但长值是特殊情况，尤其需要避免这种行为。如果发现查询受到了这个局限的影响，可以试着使用覆盖索引。更多内容请参阅第 120 页的“覆盖索引”。
- 如果在单个表里面有很多长列，那么把这些数据合并到一列中也许会更好，也可以使用 XML 文档。这让所有的数据可以共享外部存储，而不是每列都有自己的页面。
- 有时可以通过把长列存储在 BLOB 中并且使用 `COMPRESS()` 压缩它们得到性能上和空间上的好处，也可以在把它们发送给 MySQL 之前在应用程序中进行压缩。

注 5：如果操作系统交换了数据，它还是会被写到磁盘上。

注 6：这个长度足够在列上创建 255 字节的索引，即使它是 utf-8 也可以。每个 utf-8 字符可能需要 3 个字节。

为文件排序进行优化

MySQL 有两个变量可以控制如何进行文件排序。

回想一下第 176 页的“排序优化”，MySQL 有两种文件排序算法。如果需要进行排序的列的总大小加上 ORDER BY 列的大小超过了 `max_length_for_sort_data` 定义的字节，MySQL 就会使用双路排序。当任何需要的列——甚至不是用于 ORDER BY 的列——是 BLOB 或 TEXT 列的时候，也会使用双路排序。（可以使用 `SUBSTRING()` 把这些列转换为可以使用单路排序的列。）

可以通过改变 `max_length_for_sort_data` 变量的值来影响 MySQL 选择的算法。因为单路排序算法为将要排序的每一行创建了固定大小的缓冲区，VARCHAR 列的最大长度是 `max_length_for_sort_data` 规定的值，而不是排序数据的实际大小。这就是我们为什么推荐只把列设为需要的大小。

当 MySQL 不得不对 BLOB 或 TEXT 列进行排序时，它只会使用前缀并会忽略掉剩余的值。这是因为它不得不分配固定大小的结构来容纳数据并且从外部存储中将前缀拷贝回结构中。可以使用 `max_sort_length` 定义前缀应该是多大。

不幸的是，MySQL 不会真正地显示它使用的排序算法。如果增加了 `max_length_for_sort_data` 的值，并且磁盘的使用率上升、CPU 使用率下降、`sort_merge_passes` 的值比以前增加的更快，也许就该强制更多的排序使用单路排序算法。

更多关于 BLOB 和 TEXT 的内容请参阅第 84 页的“字符串类型”。

6.5.2 检测 MySQL 服务器状态变量

Inspecting MySQL Server Status Variables

按照工作负载对 MySQL 进行调优的最有生产率的方式是检查 SHOW GLOBAL STATUS 的输出，以了解哪些设置需要改变。如果你刚刚开始对服务器进行调优并且熟悉 `mysqlreport`，那么就该运行这个报告，并且检查它的输出，这会节约大量的时间。这个报告可以帮你锁定有问题的地方，并且可以使用 SHOW GLOBAL STATUS 更仔细地检查相关的变量。如果发现某些可以被改进的值，就可以对它进行调优。然后检查 `mysqladmin extended -r -i60` 产生的增量输出，以确定改变的影响。为了得到最佳的结果，应该检查变量的绝对值以及它随时间的变化。

在第 13 章中有可以使用 SHOW GLOBAL STATUS 进行检测的详细变量列表。下面仅仅显示了某些最值得检查的变量：

Aborted_clients

如果这个变量随时间增加，那么就要确定是否优雅地关闭了连接。如果不是，那就要检查网络性能，并且检查 `max_allowed_packet` 配置变量，超过了 `max_allowed_packet` 的查询会被强制地中断。

Aborted_connections

这个值应该接近于 0。不是的话，就可能是网络问题。有几个被中断的连接是正常的。例如，当某些人试着从错误的主机连接、使用了错误的用户名和密码，或者定义了无效的数据库，就会发生这样的情况。

Binlog_cache_disk_use 和 Binlog_cache_use

如果 `Binlog_cache_disk_use` 和 `Binlog_cache_use` 之间的比率很大，那么就应该增加 `binlog_cache_`

size 的值。大部分事务最好都落在二进制日志缓存里面，但是偶尔有一个发生在磁盘上也无妨。

没有非常确定方式可以减少二进制日志缓存未中 (Cache Miss)。最好的办法是增加 binlog_cache_size 并且观察缓存未中率是否下降了。一旦未中率下降到了某一个点，就不会再从加大缓存中受益。假设每秒的未中是一个，并且增加了缓存的大小，让未中减少了到每分钟一次。这已经足够好了，不可能再让它下降，即使是下降了，也不会再从中得到更多的好处，所以还不如把内存节约下来做别的事情。

Bytes_received 和 Bytes_sent

这两个值可以帮助你考察问题是由发送服务器的数据过多引起的，还是从服务器读取的数据太多引起的（注 7）。它们也许会指出代码中其他的问题，比如查询提取了超出自己需要的数据（更多内容请查阅第 161 页的“MySQL 客户端/服务器协议”）。

Com_*

应该注意不要让诸如 Com_rollback 这样不常见的变量的值超过预期值。一种检查合理的值的快捷的方式是 innotop 的命令总结模式 (Command Summary Mode)（更多关于 innotop 的内容参阅第 14 章）。

Connections

这个变量表示了连接意图的数量（不是当前连接的数量，它是 Threads_connected）。如果它的值快速增加，例如，每秒几百，那么就应该检查连接次或调整操作系统的网络堆栈（更多关于网络配置的内容见下一章）。

Created_tmp_disk_tables

如果这个值较高，有两件事情可能发生了错误：(1) 查询在选择 BLOB 或 TEXT 列的时候创建了临时表；(2) tmp_table_size 和 max_heap_table_size 可能不够大。

Created_tmp_tables

该值较高的唯一处理办法是优化查询。查询优化提示请参阅第 3 章和第 4 章。

Handler_read_rnd_next

Handler_read_rnd_next / Handler_read_rnd 显示了全表扫描的大致平均值。如果值较大，那么就应该优化架构、索引和查询。

Key_blocks_used

如果 Key_blocks_used * key_cache_block_size 的值远小于已经充分热身的服务器上的 key_buffer_size 值，那么就意味着 key_buffer_size 的值太大了，内存被浪费了。

Key_reads

要注意观察每秒钟发生的读取次数，并且将这个值和 I/O 系统进行匹配，以了解有多接近 I/O 限制。更多内容请参阅第 7 章。

注 7：即使网络容量足够大，也不要认为它不会导致性能瓶颈，网络延迟会让性能变差。

如果该值和 max_connections 相同,那么可能是 max_connections 被设置的过低或者最大负载超过了服务器的上限。但是不要假设应该增加 max_connections。它是保证服务器不会被太多负载压垮的警戒线。如果看到需求激增,那么就应该检查应用程序的行为是不是正常、服务器调优是否正确、服务器架构是否设计良好。这都比简单地增加 max_connections 要好。

Open_files

注意它不应该和 open_files_limit 的值接近。如果接近了,那么就应该增加 open_files_limit。

Open_tables 和 opened_tables

应该将该值和 table_cache 进行对照。如果每秒有太多 opened_tables,那么说明 table_cache 还不够大。表缓存没有被完全利用上时,显式的临时表也能导致 opened_tables 增加。所以说也许可以不用担心它。

Qcache_*

更多信息请参阅第 204 页的“MySQL 查询缓存”。

Select_full_join

全联接是无索引联接,它是真正的性能杀手。最好能避免全联接,即使是每分钟一次也太多了。如果联接没有索引,那么最好能优化查询和索引。

Select_full_range_join

如果该变量过高,那么就说明运行了许多使用了范围查询联接表。范围查询比较慢,同时也是一个较好的优化点。

Select_range_check

该变量记录了在联接时,对每一行数据重新检查索引的查询计划的数量,它的性能开销很大。如果该值较高或正在增加,那么就意味着一些查询没有找到好索引。

Slow_launch_threads

该变量较大说明了某些因素正在延迟联接的新线程。它说明了服务器有一些问题,但是它并没有说明真正的原因。它通常表示系统过载,导致操作系统不能给新创建的线程分配时间片。

Sort_merge_passes

该变量较大说明应该增加 sort_buffer_size,也许仅仅只是为某些查询。检查查询并且查明哪一个导致了文件排序。最好的办法是优化查询。

该变量显示了有多少表被锁住了并且导致了服务器级的锁等待(等待存储引擎锁,比如 InnoDB 的行级锁,不会使该变量增加)。如果这个值较高并且正在增加,那么就说明了严重的并发瓶颈。这时候就应该考虑下面的优化手段:使用 InnoDB 或另外的使用了行级锁定的存储引擎;手动对大表进行分区或者使用

MySQL 5.1 或以上版本的内建分区机制；优化查询；启用并发插入或者对锁定设置进行优化。

MySQL 不会显示等待时间是多长。在写本书的时候，最好的办法也许是使用微秒粒度的慢速查询日志。更多内容请参阅第 63 页的“MySQL 剖析”。

Threads_created

如果该变量较大或正在增加，那么也许就应该增加 thread_cache_size 的值。可以通过检查 threads_cached 知道有多少线程已经在缓存中了。

6.6 每连接（Per-Connection）设置调优

除非确信自己是正确的，否则就不要全局性地增加每连接设置的值。即使不需要某些缓存，它们也会一次性地被分配好。所以庞大的全局设置可能会成为巨大的浪费。相反，应该在查询需要的时候才增加设置的值。

对于平时需要保持较小值，只在需要的时候才进行增加的设置来说，最常见的例子就是 sort_buffer_size，它控制了用于文件排序的缓存大小。即使排序的数据量很小，它也会按照设置分配全部的空间，所以如果它的值大于排序需要的空间，那么就意味着浪费。

当发现查询需要较大的排序缓冲区时，就可以在查询之前提高它的值，并且在查询之后就把值恢复为 DEFAULT。下面是一个例子：

```
SET @@session.sort_buffer_size := <value>;
-- Execute the query...
SET @@session.sort_buffer_size := DEFAULT;
```

把这种代码放到函数中也许会更方便。其他的应该基于每个连接进行设置的变量是 read_buffer_size、read_rnd_buffer_size、tmp_table_size，以及 myisam_sort_buffer_size（用于修复表）。

如果保持并且恢复定制的值，可以按照下面的方式进行：

```
SET @saved_<unique_variable_name> := @@session.sort_buffer_size;
SET @@session.sort_buffer_size := <value>;
-- Execute the query...
SET @@session.sort_buffer_size := @saved_<unique_variable_name>;
```

MySQL 服务器中最弱的部分决定了其性能，它的操作系统和硬件通常也会成为限制因素。磁盘大小、可用内存、CPU 资源、网络和连接它们的组件一起决定了系统的最终容量。

前面的章节主要讨论了优化 MySQL 服务器和应用程序。这种类型的调优是很关键的，但是同时也应该考虑到硬件并且适当地配置系统。例如，如果工作负载是 I/O 密集型的，那么设计应用程序的一个目的就是最小化 MySQL 的 I/O 负载。但是在通常情况下，升级 I/O 子系统、安装更多内存或重新配置已有磁盘是更为明智的选择。

硬件变化非常迅速，所以本章不会比较不同的产品或提及某个特定的硬件，相反地，本章的目的是提供一系列的指南，以及解决硬件和操作系统瓶颈的方案。

本章开头先研究了限制 MySQL 性能的因素。最常见的是 CPU、内存及 I/O 瓶颈，但它们并不是显而易见的。我们还探讨了如何为 MySQL 服务器选择 CPU，然后考虑了如何平衡内存和磁盘资源。我们考察了不同类型的 I/O（随机 I/O 和顺序 I/O，读和写），并且解释了如何理解工作集。这些知识有助于选择适当的内存磁盘比。接下来，我们给出了为 MySQL 选择磁盘的提示，还讨论了和 RAID 优化相关的所有重要主题。在存储部分讨论了外部存储选项（比如存储区域网络），还有如何并在何时为 MySQL 数据和日志使用多个磁盘。

讨论完存储之后，接下来就是网络性能，以及如何选择操作系统和文件系统。随后介绍了线程支持，以及如何避免交换。本章最后有一些操作系统状态输出的例子。

306 7.1 什么限制了 MySQL 的性能

What Limits MySQL's Performance?

许多不同的硬件可以影响 MySQL 的性能，但是最常见的两个瓶颈是 CPU 饱和与 I/O 饱和。CPU 饱和发生在 MySQL 使用的数据能被装入内存，或者能够尽快根据需要从磁盘上读取的时候。密集的加密操作及在多个产品之间执行无索引的连接操作都是引发 CPU 饱和的例子。

从另一方面来说，I/O 饱和通常发生在需要的数据比内存多得多的时候。如果应用程序分布在网络上，查询数据相当巨大，或者要求很低的延时，瓶颈就会转移到网络上。

发现瓶颈通常不是显而易见的事情。某个区域的弱点通常会给其他的子系统带来压力，问题通常就会表现在其他子系统上面。例如，假设没有足够的内存，MySQL 就会刷写（Flush）缓存，为需要的数据腾出空间，然后再马上把数据重新读回来（读写操作都会这样）。因此内存不足就表现为 I/O 容量不够。同样地，达到饱和的内存问题会表现为 CPU 问题。事实上，在我们说应用程序有“CPU 瓶颈”或“CPU 密集”的时候，通常是指瓶颈出在计算上。稍后再详细解释这个问题。

7.2 如何为 MySQL 选择 CPU

在升级当前硬件或订购新硬件的时候，应该考虑工作负载是不是 CPU 密集的。

可以通过检查 CPU 使用率来确认负载是否是 CPU 密集的，但要注意不是只检查 CPU 总体负载有多重，而是要查明对于最重要的查询，CPU 使用率和 I/O 之间的平衡，以及注意 CPU 的负载是否均衡。可以使用 `mpstat`、`iostat` 及 `vmstat`（例子在本章最后）这些工具来查明什么因素限制了服务器的性能。

7.2.1 快 CPU 好还是多 CPU 好

对于 CPU 密集的负载，MySQL 通常从更快的 CPU 中获益（不是更多 CPU）。

但事实并不总是如此，因为这依赖于负载和 CPU 的数量。但是，当前 MySQL 的架构对多 CPU 的扩展性不好，并且 MySQL 不能在多个 CPU 上并行地运行某个查询，因此在对于单个 CPU 进行密集的查询时，CPU 速度限制了响应时间。

通常说来，我们想要的性能有两类：

低延迟（快速响应时间）

为实现这个目标，需要快速的 CPU，因为单个查询只能使用一个 CPU。

高吞吐量

如果可以同时运行很多查询，那么就可能会从多 CPU 中得益。但是，这是否能奏效还取决于许多因素。因为 MySQL 在多个 CPU 上的扩展性很差，所以通常使用更快的 CPU 更好。

如果使用了多个 CPU，并且没有并发地运行查询，MySQL 的后台线程可以为一些任务分配额外的 CPU 资源。这些任务包括清理 InnoDB 缓冲区、网络操作等。但是，这些工作比起查询来通常不算什么。如果使用双 CPU 系统来运行一个 CPU 密集的查询，那么某个 CPU 可能在 90%的时间都是空闲的。

快速的 CPU，而不是多个 CPU，会让 MySQL 复制（在下一章讨论）达到最佳工作状态。如果工作负载是 CPU 密集型的，主服务器上的并行负载能轻易地变成从服务器无法跟上的单一负载，即使从服务器比主服务器更强也不能避免这种情况。这说明 I/O 子系统，而不是 CPU，常常会成为从服务器的瓶颈。

如果工作负载是 I/O 密集型的，决定到底是采用快速的 CPU 还是多 CPU 的方式就需要搞明白查询到底做了什么。在硬件层，查询可能处于执行状态或等待状态。最常见的等待原因有几种：查询在运行队列中（进程可以运行，但是 CPU 正忙）、等待锁定，以及等待磁盘或网络。在这些情况下，通常需要更快的 CPU。（但也有例外，例如查询正在等待 InnoDB 日志缓冲互斥量，它只有在 I/O 完成之后才会被释放掉。这意味着实际需要更多的 I/O 容量。）

这说明 MySQL 在某些负载下能够使用多个 CPU。例如，假设有许多连接正在查询不同的表（因此不会竞争表锁，但对于 MyISAM 表和 Memory 表，这会成为问题），并且服务器的总吞吐量比单个查询响应要重要得多。吞吐量在这种场景下会很高，因为所有的线程可以并发地运行并且彼此不会有竞争。要再次说明的是，这在理

论上比在实际上工作得更好：不管查询读取的是不是是不一样的表，InnoDB 都有扩展性问题，并且 MyISAM 在每个键缓冲区上有全局锁。MyISAM 表上的全表扫描是查询可以并发地运行而不会引发竞争的一个例子。

308 MySQL 宣称正处于设计中的 Falcon 存储引擎可以使用至少 8 个 CPU，所以未来的 MySQL 也许能比现在更有效地使用多 CPU。但是只有时间能证明 Falcon 的真实扩展性。

7.2.2 CPU 架构

64 位架构

现在 64 位架构比几年前更加流行。MySQL 在 64 位架构上工作得很好，尽管有某些内部机制不能与 64 位系统兼容。例如，在 MySQL5.0 中，每个 MyISAM 键缓冲区都被限制在 4GB，这是由 32 位整数寻址的地址空间。（但是可以通过定义多个键缓冲区来绕过这个问题。）

为所有的新硬件选择 64 位架构是一个好主意。如果没有使用 64 位操作系统和 64 位 CPU，就不能有效地使用大量的内存。尽管一些 32 位系统能够支持大量的内存，但是系统不能像 64 位系统那样有效地使用它们，并且 MySQL 也不能很好地使用它们。

7.2.3 扩展到多个 CPU 和核心

多 CPU 系统

多 CPU 能真正派上大用场的地方是在线事务处理系统（OLTP）。这种系统通常会做很多小的操作，它们可以在多个 CPU 上运行的原因是它们来自多个连接。在这个环境下，并发会成为瓶颈。大部分网络应用都符合这个范畴。

在线事务处理服务器通常使用 InnoDB，它有一些未解决的多 CPU 问题。但是，并不只是 InnoDB 会成为瓶颈。任何共享的资源都可能成为潜在的问题。InnoDB 得到了很多关注，因为它是高并发环境下最常见的存储引擎，但是即使没有改变任何数据，MyISAM 在真正的负载下也不会表现得更好。许多并发瓶颈，比如 InnoDB 行级锁和 MyISAM 的表锁，不会在内部被优化掉，除了尽快完成工作之外没有别的优化方式，所以锁会被授权给任何等待它的因素。如果有一个锁导致了所有的等待状态，有多少 CPU 也无济于事。因此，甚至某些高并发的负载也会从更快的 CPU 中得益。

在数据库中实际有两种类型的并发问题，需要用不同的方式来解决：

逻辑并发问题

解决对应用程序可见的资源的竞争，例如表锁或行锁，通常需要的策略是改变应用程序、使用不同的存储引擎、改变服务器配置，或者使用不同的锁定提示或事务隔离层次。

309 内部并发问题

对信号量、InnoDB 缓冲池中的页面访问权的争夺，可以通过改变服务器设置、改变操作系统或使用不同的硬件来解决，但是问题可能一直会存在。在某些情况下，使用不同的存储引擎或存储引擎的补丁有助于减轻这类问题。

MySQL 能有效使用的 CPU 的数量，以及它在负载增加时的“扩展模式”依赖于负载和系统架构。“系统架构”

的意思是操作系统和硬件，而不是应用程序。CPU 架构（RISC、CISC、管道深度等）、CPU 模型及操作系统都会影响 MySQL 的扩展模式。这是评测相当重要的原因：一些系统在并发增加的时候性能依旧很好，但是另外一些系统就很糟糕。

一些系统在有多个处理器时性能可能还会下降，这是很普遍的现象。我们知道很多人从四核系统升级成八核系统后，又降级回四核系统（有的把 MySQL 强制绑定在八核中的四个核上），这是因为性能下降了。只有进行评测才能知道系统在具体的负载下会如何。

一些 MySQL 扩展性瓶颈在服务器上，而另外一些在存储引擎上。存储引擎是如何设计的至关重要，并且有时可以切换到不同的存储引擎上，从多 CPU 得到更多好处。

处理器速度的战争在某种程度上已经结束，现在 CPU 的供应商更多地集中在多核心或多线程上。未来的 CPU 可能会有上百个核心，现在四核的 CPU 已经很普遍。各个供应商的内部架构有很大不同，因此不可能对线程、CPU、内核之间的交互进行通用化。内存和总线的设计同样也非常重要。最后说来，有多核更好还是有多个物理 CPU 更好也是和架构相关的问题。

7.3 平衡内存和磁盘资源

Balancing Memory and Disk Resources

拥有大量内存的最大原因不是能够在内存中保存大量的数据，它的最终目的是可以避免磁盘 I/O，访问磁盘的速度比访问内存慢几个数量级。问题的微妙之处在于平衡内存和磁盘大小、速度、开销和其他的因素，以得到好的性能。在开始行动之前，让我们先回顾一些基本概念。

计算机内的缓存是金字塔结构，越小的越快，也越昂贵。如图 7-1 所示。

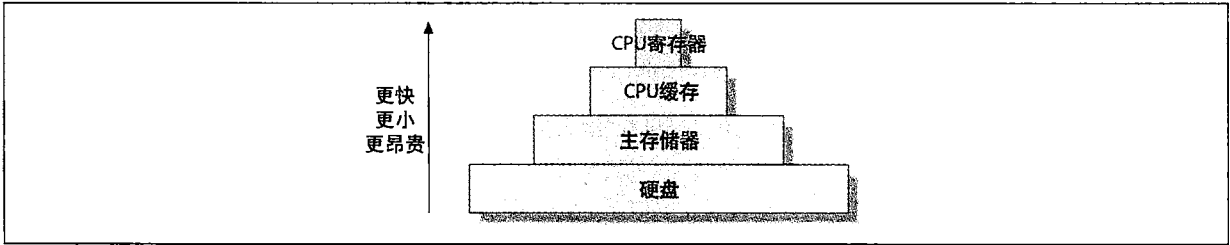


图 7-1：缓存层次

每一种缓存都应该用来存储“热”数据，以使它们可以更快地被访问到。通常的原则就是“最近使用的数据可能会很快地被再次使用”，以及“靠近最近使用的数据可能很快就会被用到。”由于时空局部访问性（Spatial and Temporal Locality of Reference），这种假设是正确的。

从程序员的观点来说，CPU 寄存器和缓存都是透明并且和架构相关的。编译器和 CPU 会管理它们。但是，程序员对主存和硬盘的区别非常清楚，而且程序处理它们的方式通常有很大的区别（注 1）。

这对于数据库服务器尤为正确，它的行为通常和我们刚才描述的假设相反。有良好设计的数据库缓存（例如 InnoDB 缓存池）通常比针对通用任务的操作系统的缓存效率更高。数据库缓存更了解自己的数据需要，并且它有特定的逻辑来满足这种需要。同样地，访问数据库缓存中的数据不需要系统调用。

注 1：但是，程序可能会依赖于操作系统，把理论上应该在磁盘上的数据缓存到内存中。MyISAM 就是这么做的。

这些特定的缓存需求是平衡缓存机制来满足数据库服务器特定访问模式的原因，因为寄存器和芯片上的缓存都不是用户可以配置的，唯一能改变的就是主存和磁盘。

7.3.1 随机 I/O 和顺序 I/O

Random Versus Sequential I/O

数据库服务器同时使用了随机 I/O 和顺序 I/O，随机 I/O 从缓存中得益是最多的。可以用一个典型的混合负载来说明这个问题，该负载由单行查找和多行范围扫描组成。这种“热”数据的典型模式是随机分布，因此缓存这些数据就可以避免昂贵的磁盘搜索。相反地，顺序读取通常只会扫描一遍数据，除非它能被全部装入内存，否则就没必要缓存它。

311 顺序读取不会从缓存中获益的另外一个原因就是它比随机读取快。这有两个原因：

顺序 I/O 比随机 I/O 快

内存和磁盘中的顺序操作都比随机操作快。假设磁盘每秒能做 100 次随机 I/O 操作，并且每秒能读取 50MB 顺序数据（现在大部分普通硬盘都能达到这个要求）。假设每行数据是 100 字节，那么每秒就能随机读取 100 行，但是对于顺序读取，每秒就能读取 500 000 行，这是 5 000 倍，或者是几个数量级的差别。因此，在这种场景下，随机 I/O 会从缓存中受益更多。

顺序访问内存中的数据也比随机访问快。现在的内存芯片每秒能够随机访问大约 250 000 行 100 字节的数据。要注意到访问内存中的数据比访问磁盘数据快 2 500 倍，而内存中的顺序访问只比随机访问快 10 倍。

存储引擎能更快地执行顺序读取

随机读取通常意味着存储引擎需要做索引操作。（也有例外，但是对于 InnoDB 和 MyISAM 是这样的。）这通常需要遍历 B 树数据结构并且对值执行比较操作。相反地，顺序读取通常只需要遍历简单的数据结构，比如链接列表。这会少很多工作，所以顺序读取还是更快。

可以通过缓存顺序数据来减少工作量，但是缓存随机数据好处更大。换句话说，对于随机读取 I/O 问题，添加内存是最佳选择。

7.3.2 缓存、读取和写入

Caching, Reads, and Writes

如果有足够的内存，可以完全不进行磁盘读取。如果所有的数据都在内存中，那么每次读取都会发生在缓存中。仍然会有逻辑读取，却不会有物理读取。但是写入不同。写入可以像读取一样，发生在内存上，可它早晚得写到磁盘上。换句话说，缓存可能会延迟写入，但是不能消除写入。

事实上，除了运行延迟写入之外，缓存还能用两种重要的方式把它们集中在一起：

多次写入，一次刷写

内存中的数据在没有被写入到磁盘中时，可以被修改很多次。当数据最终被刷写到磁盘上，所有的修改就变成了永久性的。例如，许多语句都可以更新内存中的计数器。

如果计数器被增加了 100 次，然后写入磁盘，那么 100 次修改就被综合到了一次写入中。

I/O 合并

许多不同的数据片段都可以在内存中被修改，并且修改可以被集中到一起，那么就能在一次磁盘操作中完成物理写入动作。

这是许多事务系统使用预写（Write-ahead）日志策略的原因。预写日志使改变发生在内存中，而不是直接写到磁盘上。直接写入通常是随机 I/O 操作，速度也很慢。预写把改变写入顺序日志文件中，这通常快得多。后台线程能够随后把修改后的页面写到磁盘上，这样就可以优化写入。

写入能极大地从缓存中获益，因为它把随机 I/O 转换为顺序 I/O。异步（缓存）写入通常由操作系统处理，并且是批次进行的，所以它们能够更优化地被刷写到磁盘上。同步（非缓存）写入在它们完成之前就要被写入磁盘，这是为什么它们能从独立冗余磁盘阵列（RAID，Redundant Array of Independent Disk）控制器的缓冲中获益的原因（稍后讨论 RAID）。

7.3.3 你的工作集是什么

What's Your Working Set?

每个应用程序都有“工作集”数据，也就是真正用于工作的数据。数据库也有很多不在工作集中的数据。可以把数据库想象成一个有很多抽屉的桌子，工作集就是桌面上用于实际工作的文件。在这个类比中，桌面就是主存，抽屉就是硬盘。

正如不是桌面上的每一张纸都要用于工作，没有必要把所有数据都放在内存中，只需要把工作集放进去就可以了。

工作集的大小和应用程序非常相关。对于某些应用程序，工作集大小也许只是总数据的 1%，然而其他程序可能是 100%。当工作集不能装入内存中的时候，数据库服务器就要在内存和磁盘之间搬动数据，以完成工作。这是为什么内存短缺会表现为 I/O 问题。有时没办法把整个工作集装入到内存中，实际上有时也不想这么做（例如，如果应用程序需要大量的顺序 I/O）。应用程序架构能够解决很多工作集是否和内存相匹配的问题。

“工作集”是一个有歧义的术语。例如，你也许每小时只需要访问 1% 的数据，但是在 24 小时内，这意味着要访问 20% 的数据。工作集在这种条件下是什么意思？从有多少数据需要进行缓存这个方面来考虑工作集可能更有帮助，因此负载大部分都是 CPU 密集的。如果不能缓存足够的数据，工作集就不能放到内存中。

工作集和缓存单元

工作集包含了数据和索引，并且应该用缓存单元来进行统计。一个缓存单元是存储引擎能接受的最小数据单元。

存储引擎之间缓存单元的大小各不相同，所以工作集大小也会不一样。例如，InnoDB 的页面始终为 16KB。如果执行单行查找，并且 InnoDB 不得不到磁盘上去取数据，它就会把包含该行的整个页面读取到缓冲池中缓存起来。这可能比较浪费。假设要随机访问的行的行大小是 100 字节。InnoDB 将会在缓冲池中为它们使用大量的额外内存，因为它要为每一行读取 16KB 的页面。工作集也包含了索引，InnoDB 也将会读取并缓存查找数据所需的部分索引树。InnoDB 的索引页面也是 16KB，这意味着为访问单行长度为 100 字节的数据，就需要存储 32KB（也许更多，这取决于索引树的深度）的数据。因此，缓存单元是好的聚集索引在 InnoDB 中如此重要的另外一个原因。聚集索引不仅可以优化磁盘访问，还有助于在同一页面上保持相同的数据，这样就可以把工作集

装入内存中。

相反地，Falcon 引擎的缓存单元是行，而不是页面。因此 Falcon 在缓存小的、随机访问的、分布很宽的行上效率会较高。还是用前面办公桌的例子，InnoDB 要求把这个文件夹拿出来，然后在里面找到自己需要的文件。如果没有聚集索引（或者聚集索引设计得很糟糕），效率会很低下。从另一方面来说，Falcon 使你可以直接从抽屉中拿出自己需要的文件，而不用移动整个文件夹。

这两种方式都有优点和缺点。例如，InnoDB 把整个 16KB 页面保存到了内存中，如果未来需要访问同一页面中的某一行，就会比较快。Falcon 同时有行缓存和页缓存，它有综合的好处：页面缓存减少了磁盘访问，并且行缓存能高效地使用内存。但是双重缓存有一些浪费，因为它使某些数据在内存中存储了两次，这被称为“双缓冲”。

在理论上，两种策略对于特定的工作负载都可以更快或更慢，具体选择总是依赖于存储引擎进行的工作。

7.3.4 找到有效的内存磁盘比

好的内存磁盘比最好能通过实验或评测来发现。如果可以把所有东西装入内存中，那就不用考虑这个问题。但是大部分时候都做不到这一点，所以需要有一部分数据进行评测，看看发生了什么。目标是找到一个可接受的缓存未命中率。缓存未命中是指请求的数据没有被缓存中内存中，服务器不得不从磁盘上去取。

缓存未命中率通常体现在使用了多少 CPU 上面，因此最好的衡量缓存未命中率的方式是查看 CPU 使用率。例如，有 90%的时间在使用缓存，并且有 10%的时间等待 I/O，那么缓存未命中率就处于良好状态。

让我们考虑工作集如何影响缓存未命中率。重要的是要认识到缓存未命中率不只是一个数字，它实际是一种统计分布，并且缓存未命中率和分布并不是线性关系。例如，如果有 10GB 内存，并且缓存未命中率是 10%，那么你有可能会想再增加 11%的内存（注 2）就会把缓存未命中率降到 0。但是在现实中，由缓存单元带来的低效率也许意味着需要 50GB 的内存才能得到 1%的缓存未命中率。有时甚至在缓存单元完美匹配的时候，理论估计也可能是错误的。诸如数据访问模式等因素会把事情变得更加复杂，1%的缓存未命中率可能会需要 500GB 内存！

在优化的时候很容易走入歧途。例如，10%的未命中率已经导致了 80%的 CPU 使用，这非常好。假设增加内存，并且把未命中率降低到了 5%。粗略估计下来，此时会把另外 6%的数据交给 CPU。再进行一次粗略估计，此时 CPU 的使用率被增加到了 84.8%。但是，这不是很大的胜利。想象一下需要购买多少内存。并且在现实中，由于内存和磁盘访问的速度区别、CPU 处理数据，以及其他的因素，将缓存未命中率降低到 5%可能根本就不会大幅度改变 CPU 使用率。

这是为什么要为可接受的缓存未命中率，而不是 0%未命中率而努力的原因。并没有一个确定的数字，因为“可接受”依赖于应用程序和工作负载。一些应用程序在 1%的缓存未命中率时已经工作得很好，但是另外的应用也许会需要 0.01%。（良好的“缓存未命中率”和“工作集”一样，是模糊的概念，并且实际上有很多方法考量未命中率。）

最佳的内存磁盘比也取决于系统中其他的组件。假设系统有 16GB 内存，20GB 数据，还有很多未使用的磁盘空间。系统在 80%CPU 使用率下工作得很好，如果想在系统中使用双倍的数据，并且得到同样的性能，那么你有

注 2：正确的数字是 11%，不是 10%。10%的未命中率是 90%的命中率，所以应该用 10GB 除以 90%，就等于 11.111GB。

可能打算增加一倍 CPU 和内存。但是，即使系统中的每个组件都能随着负载完美地扩展（只是不可能的假设），这也不可能。20GB 数据可能已经使用了系统组件 50% 以上的容量，例如，它可能已经达到了系统每秒最大 I/O 操作的 80%。它就不能处理双倍的负载。因此，最佳的内存磁盘比取决于系统中最弱的部分。

7.3.5 选择硬盘

如果不能把足够多数据放到内存里面，比如估计需要 500GB 内存才能在当前 I/O 系统下让 CPU 达到满负荷，那么就应该考虑更强大的 I/O 子系统，同时应该让应用程序可以处理 I/O 等待。

这可能有点违背直觉。我们刚刚才说过更多内存能减少 I/O 系统的压力并且减少 I/O 等待。如果增加内存能解决问题，为什么还要加强 I/O 子系统呢？答案在于各个因素之间的平衡，例如读写的数量、每个 I/O 操作的大小及每秒发生操作的数量。例如，如果需要快速的日志写入，那么就不能通过增加内存数量来减少写入的次数。在这种情况下，投资在有备用电池的写入缓存的 I/O 系统可能是更好的选择。

从普通的硬盘上读取数据分为以下 3 个步骤：

1. 将磁头移动到磁盘表面的正确位置；
2. 等待磁盘旋转，需要的数据会移动到磁头下面；
3. 磁盘继续旋转，直到所有需要的数据都经过磁头。

磁盘在做这些动作的时候到底有多快可以归结为两个因素：访问时间（第 1 步和第 2 步）及传输速度。这两个因素也叫延迟和吞吐量。到底是需要快的访问时间，还是快的传输速度，或者是两者都要，取决于运行的查询的类型。从完成一次磁盘读取需要的总时间上来说，第 1 步和第 2 步是进行短时间的随机查找，然而大部分时间都是由第 3 步决定的。

一些其他的因素也会影响磁盘选择，到底哪些重要取决于应用程序。让我们想象下一个诸如流行的新闻网站这样的在线应用，它有许多小而随机的读取。那么就应该考虑下面的因素：

存储容量

这对在线应用基本不会称为问题，因为现在的磁盘通常都比需要的大。如果不是，那么把多个磁盘搭建成 RAID 是标准方式（注 3）。

传输速度

现在的磁盘通常传输速度都很快。到底有多快依赖于转速、磁盘表面数据的密度及磁盘和主机之间的接口（现在很多磁盘读取数据的速度都大于接口的速度）。不管怎么说，传输速度对在线应用通常都不是限制因素，因为它们通常都是进行大量的小而随机的读取。

访问时间

这通常决定了随机查找的速度，所以应该使用访问时间较快的磁盘。

注 3：有趣的是，一些人特意购买大容量硬盘，但是只使用 20%~30% 的空间。这增加了数据本地性并且降低了寻址时间，有时可以弥补高价格。

转速

现在通常的转速是每分钟 7 200 转、10 000 转或 15 000 转。转速可以决定随机查找的速度和顺序读取的速度。

物理尺寸

在其他因素一样的情况下，物理尺寸也会造成差别。较小的磁盘，移动磁头的时间也会较少。服务器级别的 2.5 英寸磁盘通常比同级别的大尺寸磁盘要快。它们使用的电力也较少，并且更容易安装到机箱中。

磁盘计算变化很快，所以建议也会很快过时。例如，固态硬盘在写作本书的时候正是热门概念。它们与普通硬盘有很大不同，但是，它们也非常昂贵并且还没有被广泛地使用。我们知道一些工程已经成功地使用了它们，但是我们现在没有足够的经验给出对它的任何建议。

和 CPU 一样，MySQL 扩展到多个磁盘依赖于存储引擎和工作负载。InnoDB 通常可以扩展到 10 到 20 个硬盘驱动器。但是 MyISAM 的表锁限制了写入扩展性，因此写入负担很重的负载也许不会从多个磁盘受益。操作系统缓冲和并行后台写入在一定程度上有帮助，但是 MyISAM 的写入扩展性不如 InnoDB。

和 CPU 一样，多个磁盘未必都是好事。一些要求低延迟的应用程序需要更快的驱动器，而不是更多的磁盘。例如，复制通常在磁盘速度较快的时候性能更好，因为从服务器上的更新是单线程的。

317 为了确定负载是否可以从多个磁盘获益，可以查询 `iostat` 以了解磁盘是如何加载的。大量的尚未处理的请求意味着工作负载也许可以更有效地使用磁盘。本章结尾有一些 `iostat` 的例子。

7.4 为从服务器选择硬件

Choosing Hardware for a Slave

为复制从服务器选择硬件通常和为主服务器选择硬件一样，尽管有一些区别。如果正在计划使用从服务器以实现故障转移，那么它至少就要和主服务器一样强大。并且无论从服务器是否会取代主服务器，都必须能运行主服务器上的所有写入动作，由于有某些缺陷，它必须顺序地执行它们。（更多内容请参阅下一章）

从服务器的主要考量是开销：你需要为从服务器花像主服务器那么多钱吗？能够通过配置从服务器得到更多的性能吗？

这要依情况而定。如果从服务器用于待命，那么主服务器和从服务器就应该有一样的配置和硬件。但是如果使用从服务器仅仅是为了增加总的读取容量，那么就可以在从服务器上采用一些捷径。可以在从服务器上使用不同的存储引擎，例如，一些人使用更便宜的硬件，或者用 RAID 0 代替 RAID 5 或 RAID 10。还可以禁用一些保证连续性和持续性的因素，以让从服务器做更少的工作。更多内容请查阅第 281 页“MySQL I/O 行为调优。”

大体上说，这些方法能够让开销更有效，但是在小的方面会使事情更复杂。

7.5 RAID 性能优化

RAID Performance Optimization

存储引擎通常把数据或索引保持到一个单个的文件中，它意味着对于存储大量的数据，RAID 通常是最可行的

选择（注 4）。RAID 对冗余、存储大小、缓存和速度都有帮助。但是和其他的优化方式一样，RAID 的配置有很多变化，选择适合自己需要的配置是很重要的。

这儿不会讨论每一种 RAID 级别，也不会讨论不同类别的 RAID 存储数据上具体的差别。这方面的材料可以从数据或网络上获取（注 5）。相反地，这儿讨论的是如何配置 RAID，以使其满足数据库服务器的需求。最重要的 RAID 级别是：

RAID 0

RAID 0 是最便宜、效率最高的 RAID 配置，至少在对开销和性能进行最简单的衡量下是这样的（如果把数据恢复这些因素考虑在内，它就显得比较昂贵了）。由于它没有提供冗余，所以我们只建议你在不重要的服务器上使用它，例如从服务器或可任意使用的服务器。一个典型的场景是从服务器可以轻易地从其他服务器克隆过来。

再强调一下，RAID 0 没有提供任何冗余，即使 RAID 的第一个字母就是冗余的意思。事实上，RAID 0 阵列失效的概率高于单个磁盘失效的概率，而不是低于！

RAID 1

在很多情况下，RAID 1 提供了很好的读取性能，并且它复制了数据，所以有良好的冗余性。RAID 1 在读取上比 RAID 0 要快一点。对于处理日志和类似的负载的服务器，它是不错的选择，因为顺序写入基本不需要磁盘有很好的性能（随机写入正好相反，它可以从并行化得益）。对于需要冗余，但却只有两个硬盘的低档服务器，它是不错的选择。

RAID 0 和 RAID 1 都很简单，并且它们可以以软件的方式很好地运行。大部分操作系统都能轻易地创建软件形式的 RAID 0 和 RAID 1。

RAID 5

RAID 5 有点让人害怕，但是它对于某些应用是不可避免的选择，这也许是由于价格，也可能是因为服务器中实际能安装的硬盘数量。它把数据分布在了许多磁盘上，并且使用了奇偶校验的数据块，如果一个磁盘失效了，那么数据可以从其他磁盘上重新构造出来。从每存储单元的开销来说，它是最经济的有冗余的配置，因为出问题时仅仅只失去了一块磁盘的存储空间。

RAID 5 的随机写入代价较高，因为它们需要两次写入和两次对奇偶校验数据块的 RAID 操作。如果是顺序写入或有很多磁盘，那么性能会好一些。从另一方面来说，顺序和随机读取的性能都还不错。就大部分工作负载而言，RAID 5 对于数据卷、数据和日志都是可接受的选择。

RAID 5 最大的性能开销出现在一个磁盘失效的时候，因为数据要通过读取所有磁盘的方式重建出来。这严重影响了性能。如果想在重建数据的过程中保持服务器在线，那么重建和服务器的性能都不可能好。另外一个性能开销包括有限的扩展性，这是由于奇偶校验数据块和缓存问题导致的。RAID 5 不能很好地扩展到 10 块以上的硬盘。RAID 5 的性能严重依赖于 RAID 控制器的缓存，它有可能和服务器的需求发生冲

注 4：分区（参看第 5 章）是另外一个好办法，因为它把文件分成了许多小文件，它们可以放在不同的设备上。但是和分区相比，RAID 对非常巨大的数据是简单的解决方案。它不需要手动平衡负载，或者在负载分布发生变化时进行干预。它还提供了冗余，分区没办法提供这个功能。

注 5：维基百科 (<http://en.wikipedia.org/wiki/RAID>) 和 AC&NC (http://www.acnc.com/04_00.html) 都提供了学习 RAID 的不错资料。

突。稍后我们再来讨论缓存问题。

另外一个减轻该问题的因素是 RAID 5 非常普遍。由于这个原因，RAID 控制器通常都为 RAID 5 进行了高度优化，尽管有理论上的限制，很好地使用了缓存的智能控制器在某些负载下能达到 RAID 10 的性能。这也许反应了 RAID 10 控制器并没有被高度优化，但是不管什么原因，这就是我们见到的情况。

RAID 10

RAID 10 是存储数据的好选择。它也被称为镜像阵列条带。它在读写方面的扩展性都不错。相对 RAID 5 来说，它速度快，并且容易重建。它也能通过软件的方式来实现。

当一个磁盘失效的时候，性能可能会受到很大的影响，因为条带会成为瓶颈。根据不同的负载，性能可能会下降到 50%。对于 RAID 控制器来说，要注意的是在 RAID 10 的实现中使用了“连接镜像”。这种方式不够优化，因为它失去了条带性。在这种方式下，最常用的数据只是被放在了一对磁盘上，而不是分布在多个磁盘上，这样性能就很差。

RAID 50

RAID 50 由分带的 RAID 5 阵列组成，它在 RAID 5 的经济性和 RAID 10 的性能之间得到了不错的折中。对于非常大的数据集，比如数据仓库或极大的 OLTP 系统，它得到了广泛的使用。

表 7-1 总结了各种 RAID 的配置。

表 7-1：RAID 级别的比较

| 级别 | 特点 | 冗余 | 需要的磁盘 | 较快读取 | 较快写入 |
|---------|------------------|----|----------|------|---------|
| RAID 0 | 便宜、快速、危险 | 否 | N | 是 | 是 |
| RAID 1 | 快速读取、简单、安全 | 是 | 通常为 2 | 是 | 否 |
| RAID 5 | 在安全、速度和开销之间进行了折中 | 是 | $N + 1$ | 是 | 和其他因素相关 |
| RAID 10 | 昂贵、快速、安全 | 是 | 2N | 是 | 是 |
| RAID 50 | 适用于极多的数据 | 是 | $2(N+1)$ | 是 | 是 |

7.5.1 RAID 失效、恢复和监视

RAID Failure, Recovery, and Monitoring

RAID 配置（除了 RAID 0）提供了冗余性。这很重要，但是这也很容易让人低估磁盘同时失效的可能性。不应该把 RAID 看成保证数据安全的强有力手段。

RAID 不能消除，甚至不能减少备份的需要。当问题发生的时候，恢复的时间依赖于控制器、RAID 级别、阵列大小、磁盘速度，以及是否需要在恢复的时候保持服务器在线。

磁盘有可能同时失效。例如，电流冲击或过热很容易让两个或更多的磁盘损坏。但是，更常见的情况是两个磁盘在很短的时间内相继损坏。许多这样的问题都不会被注意到。一个常见的例子就是保存很少被访问的数据的磁盘损坏，这个问题可能在几个月内都不会被发现，直到需要读取数据，或者其他磁盘损坏了，控制器需要用到该磁盘上的数据来进行恢复的时候，才会发现问题。磁盘越大，出问题的几率就越高。

这就是为什么监视 RAID 阵列很重要的原因。大部分控制器提供了一些软件来报告阵列的状态，你需要跟踪这些状态，如果不这样，就有可能忽略磁盘损坏。在第二块磁盘损坏的时候，可能就没有机会恢复数据了，因为这时候已经太晚了。

可以通过常规的阵列连续性检查来解决这个问题。后台巡逻 (Background Patrol Read) 是某些控制器提供的功能，它能在服务器在线的时候检查数据损坏并且尝试修复它，这项功能防止这样的问题。和恢复一样，极大的阵列检查的速度会较慢，所以要确保创建阵列的时候就要计划好。

还可以添加多余的驱动器，它不会被使用，只是在恢复的时候作为支持。如果自己的每一台服务器都很重要，这会是个好主意。如果服务器只有几个磁盘，那么一个空闲的磁盘就显得有点奢侈，但是如果有许多磁盘，那么就一定要有备用磁盘。因为磁盘越多，失效的概率就越大。

7.5.2 平衡硬件 RAID 和软件 RAID

操作系统、文件系统及磁盘之间的交互是复杂的，缺陷、局限或错误配置都能降低理论上可以达到的性能。

如果有 10 块硬盘，最理想的情况是它们可以并行地响应 10 个请求，但是有时文件系统、操作系统或 RAID 控制器会将请求进行串行。解决这个问题一个办法是尝试不同的 RAID 配置。例如，如果有 10 块硬盘并且打算为冗余和性能使用镜像，那么就可以尝试几种配置方式：

- 将 5 对镜像磁盘配置为一个 RAID 10 卷。操作系统只会看见一个磁盘，RAID 控制器隐藏了背后的 10 个磁盘。
- 在 RAID 控制器中配置为 5 个 RAID 1，那么操作系统就会看见 5 个磁盘卷。
- 在 RAID 控制器中配置为 5 个 RAID 1，然后使用软件 RAID 0 把它变为一个逻辑卷，这样就利用硬件和软件结合的方式实现了 RAID 10。

那个方案最好？这取决于系统中组件的交互方式。这些配置性能可能一样，也可能不一样。

我们注意到在不同配置中的串行问题。一个例子（发生在一个已经过时的 GNU/Linux 版本上）就是使用了 ext3 文件系统，并把 InnoDB 的 `innodb_flush_method` 设置为了 `O_DIRECT`。这表现为文件系统中索引节点级的锁定，所以一次只有一个 I/O 请求被发送到文件。在这种情况下，每个文件都被串行化了。这个问题在随后的版本中被修复了。

在另外一种情况下，发送到每一个设备的请求被串行化到了一个由 10 个磁盘组成的 RAID 10 卷上，这时文件系统是 ReiserFS，并且 InnoDB 的 `innodb_file_per_table` 是开启的。在硬件 RAID 1 的基础上加上软件 RAID 0，把吞吐量提高了 5 倍，因为这时存储系统认为有 5 个硬盘，而不是一个。这个问题也是由缺陷引起的，并且也被修复了。但是要知道这类问题有可能会发生。

串行化可以发生在软件或硬件堆栈的每一个层次上。如果发生了这样的问题，就应该试着改变文件系统、升级内核，把更多设备暴露给操作系统，或者混合使用不同的软件或硬件 RAID。也可以用 `iostat` 来检查设备的并行性并确保它真正地执行了并发 I/O。（更多内容请参阅第 338 页“如何读取 `iostat` 输出”）。

最后，不要忘了评测。这可以帮你验证是否得到了想要的性能。例如，如果一个硬盘每秒能做 200 次随机读写，那么一个由 8 个硬盘组成的 RAID 10 卷每秒进行的随机读写就应该接近于 1600 次。如果看到了小得多的

数目，比如每秒 500 次，那么就应该进行研究。要保证使用和 MySQL 一样的方式测试 I/O 子系统，例如，对于没有开启 `innodb_file_per_table` 的 InnoDB 来说，就要使用 `O_DIRECT` 标记并且对单个文件测试 I/O 性能。SysBench 是很好的测试工具。（更多关于评测的内容请参阅第 2 章）

7.5.3 RAID 配置和缓存

RAID Configuration and Caching

可以在机器启动的时候使用机器的设置工具来配置 RAID 控制器。大部分控制器都提供了很多选项，我们这儿只讲解其中两个，一个是条带型阵列的块大小（Chunk Size），另外一个控制器缓存（On-controller Cache）（也叫 RAID 缓存）。

RAID 条带块大小

适当的条带块大小与负载和硬件相关。在理论上，对于随机 I/O，最好能有较大的条带块大小，因为这意味着更多的读取可以在一个驱动器上完成。

为了明白其中缘由，可考虑一下负载的典型 I/O 操作的大小。如果块大小满足最小要求，并且数据没有在各个块之间分布，那么就只需要一个驱动器参与到读取中。但是如果块的大小小于将要读取的数据数量，那么就肯定需要从多个驱动器上读取。

理论上就这些。在实际情况中，许多 RAID 控制器在大型块的情况下都工作得不好。例如，控制器可能会把块大小当成缓存单元，这将造成浪费。控制器也有可能匹配块大小、缓存大小及读取单元的大小（在一次操作中读取数据的数量）。如果读取单元过大，缓存效率就会降低，并且它会读取比真正需要的更多的数据，即使对于很小的请求也不例外。

同样，在实际中难以知道特定的数据是否会跨越多个驱动器。即使块大小是 16KB，它和 InnoDB 的页面大小相匹配，也不能肯定所有的读取都能和 16KB 的边界对齐。文件系统有可能会把文件弄成碎片，并且它的大小通常是 4KB。一些文件系统更智能，但是并不能总是指望这一点。

323 RAID 缓存

RAID 缓存是安装在 RAID 控制器中的少量（相对说来）的物理内存。它可以缓冲在磁盘和主机系统之间来往的数据。下面是一些 RAID 卡应该使用缓存的原因。

缓存读取

在控制器从磁盘读取了数据并且把它们发送到主机系统之后，缓存就会把数据保存起来。以后再次请求这个数据的时候就不用从磁盘中读取。

这种使用缓存的方式很差劲。为什么呢？因为操作系统和数据块服务器都是自身的缓存，而且要大得多。如果在自身的缓存中找到了数据，那么就不会使用 RAID 的缓存，但是如果找不到，在 RAID 缓存中找到数据的机会也会微乎其微。RAID 缓存是如此之小，所以基本可以肯定老的数据会被清理掉，然后填上新的数据。不管从哪个方面看，在 RAID 缓存中进行缓存读取都是浪费内存的行为。

缓存预读数据

如果 RAID 控制器发现对数据正在进行顺序请求，它可能就会进行预读，也就是说，它会预测并提取将会用到的数据。但是需要有地方来保存这些数据，直到收到请求，这时可以使用 RAID 缓存。这种方式对性能的影响可能是多种多样的，并且应该对实际情况进行检查。预读操作在数据块服务器进行自身智能预读（就像 InnoDB 所做的那样）的时候也许没什么帮助，并且它还可能干扰同步写入的所有重要的缓冲。

缓存写入

RAID 控制器能够在自身的缓存中缓冲写入操作，并把它们安排到稍晚的时候。这样做有两个优势：首先，它能更快地给主机系统返回“成功”信号，而不用等到实际写入完成；其次，它能将写入累积起来，并且更有效率地执行。

内部操作

一些 RAID 操作非常复杂，特别是 RAID 5 写入，它需要计算用于重建数据的奇偶校验位。控制器需要一些内存来进行这种内部操作。

这是 RAID 5 在一些控制器上性能很差的一个原因，为了得到好的性能，它需要读取大量的数据到缓存中。一些控制器不能平衡缓存写入与计算奇偶校验位需要的缓存。

通常说来，RAID 控制器的内存是很稀缺的资源，需要明智地使用。把它用于读取通常是一种浪费，但是用于写入是加速 I/O 性能的重要方式。许多控制器都可以选择如何分配内存。例如，可以选择用于读取的内存数量和用于写入的内存数量。对于 RAID 0、RAID 1 和 RAID 10，应该把 100% 的内存都用于写入。对于 RAID 5，应该保留一些内存用于内部操作。这通常是不错的建议，但是并不是金科玉律，不同的 RAID 卡需要不同的配置。

当把 RAID 缓存用于写入的时候，许多控制器都能配置延迟写入的时间（1 秒、5 秒等）。更长的延迟意味着更多的写入会被安排到一起执行，它的缺点就是写入会更有“冲击性”。这并不是坏事，除非缓存已满并打算执行写入时，应用程序又发送了大量的写入请求。如果没有足够的空间给写入缓存，它就不得不等待。较短的延迟意味着更多的写入操作和更低的效率，但是它降低写入尖峰，并且能为应用程序的突发请求留出一些空间。（这儿我们进行了一些简化，控制器通常都有复杂的、供应商特有的平衡算法，所以我们只是描述了基本原则。）

写入缓存对于同步写入非常有帮助，例如在事务日志中发起 `fsync()` 调用，以及在 `sync_binlog` 开启的情况下创建二进制日志。但是如果控制器没有电池备份单元（BBU Battery Backup Unit），就不要开启这个选项，它有可能在失去电力的时候破坏数据块，甚至事务文件系统。但是如果有 BBU，启用写入缓存能提高 20 倍性能，在负载进行大量的日志刷写，比如在事务提交的同时刷写事务日志时还会更多。

最后的考虑就是许多硬盘都有自己的写入缓存，它们会“假冒” `fsync()` 操作，向控制器撒谎说数据已经被写入到磁盘上了。直接连接的硬盘驱动器（不是连接到 RAID 控制器）的缓存有时可以被操作系统管理。但这也并不总是成功的。这些缓存执行 `fsync()` 后就会被清空，并且同步 I/O 会绕过它们，但是要再次强调的是，硬盘会撒谎。我们要么保证它能在执行 `fsync()` 的时候被清空，要么就禁用它们，因为它们没有备用电池。操作系统或 RAID 固件不能正确地管理硬盘已经造成了很多数据丢失的例子。

出于这样或那样的原因，在安装新硬件的时候进行真正的崩溃测试总是好主意（从墙上拔掉电源插头），这通常是发现一些微妙的配置问题和硬盘的奇异行为的唯一方式。可以到 <http://brad.livejournal.com/2116715.html> 下载一个方便的脚本来进行这个测试。

325 如果真地依赖于 RAID 控制器的 BBU, 要确保测试 BBU 的时候电源断开的时间能达到要求, 因为一些数据在断开电源的时候不能被保持应有的时间。再强调一次, 一个错误就能导致整个存储链上的组件失效。

7.6 存储区域网络和网络附加存储

存储区域网络 (SAN) 和网络附加存储 (NAS) 是两种既不同, 又相互关联的外部文件存储设备。SAN 直接连接服务器, 向服务器暴露了数据块级的接口, 但是 NAS 设备向服务器暴露了基于文件的协议, 比如 NFS 或 SMB。SAN 通常通过光纤通道协议 (FCP Fiber Channel Protocol) 或 iSCSI 连接到服务器, 但是 NAS 通过标准网络连接和服务器进行连接。

7.6.1 存储区域网络

使用 SAN 的好处包括更灵活的存储管理和存储的扩展性。许多 SAN 解决方案都有特殊的功能, 比如快照和支持持续集成备份。它们让服务器可以访问大量的硬盘——通常超过 50 个——并且通常有很大的智能缓存以缓冲写入。它暴露给服务器的数据块级的接口是逻辑单元号 (LUN) 或虚拟卷。许多 SAN 也允许多个节点被“聚集”在一起, 以加强性能。

尽管 SAN 在有很多并发请求并且需要高吞吐量时工作得很好, 但是也不能期望它有魔法。SAN 最终还是硬盘的集合, 每秒进行的 I/O 操作也是有限的, 并且由于 SAN 在服务器的外部并且有自身的处理过程, 它增加了每个 I/O 请求的延迟。这种额外的延迟使 SAN 需要很高的同步 I/O 性能时缺乏效率, 因此将事务日志保存在 SAN 上通常不如将它放在 RAID 控制器上。

通常说来, 对于同样数量的类似硬盘, 直接连接会比在 SAN 上使用 LUN 快。通过 LUN 共享硬盘使性能分析更加复杂, 因为 LUN 造成的影响难以进行量化。当把硬盘放在不同的 LUN 上时, 它的影响难以察觉, 但是有时还是可以察觉到。比如, 在使用 iSCSI 的时候, 有可能会看见各个网络部分之间的冲突。SAN 内部的软件也有自身的局限, 它能使实际的性能和理论性能或预期性能有所不同。

SAN 有一个很大的缺点, 它的开销通常比直接连接的存储设备的开销大得多 (尤其对于内部存储设备而言)。

326 大部分网络应用不使用 SAN, 但是它在所谓的企业应用中使用非常广泛。下面是一些原因:

- 企业应用通常在预算方面的限制更少, 但是 SAN 对于网页应用过于“奢华”了。
- 企业通常会允许许多程序, 或者一个程序的许多实例, 并且需求增长也不可预测。SAN 意味着购买了大量的存储空间, 也拥有了共享和按需增长的能力。
- SAN 的大型缓存可以吸收写入请求的尖峰, 还能为“热”数据提供更快访问速度。SAN 还能在大量的硬盘上平衡负载。所有这些都对垂直扩展的集群应用程序有帮助, 但是对网页应用不会有太大的作用。网页应用不会在大规模写入之后就进入活动变少的状态, 大部分网页应用都会持续地写入大量的数据, 因此缓存写入不会有太大的作用。同样也不需要读取缓存, 因为数据有自己的 (大而且智能) 缓存。构建大型网页应用的常见而成功的策略就是使用应用分区, 所以网页应用已经可以在大量的硬盘上平衡负载了。

7.6.2 网络附加存储

Network-Attached Storage

NAS 设备从本质上说是完全全的文件服务器，通常有网页界面，而不是物理鼠标、显示器和键盘。它能以经济而安全的方式提供大量的存储空间，并且通常在 RAID 阵列的基础上提供冗余。

但是，NAS 设备并不快，因为它们是通过网络加载的。它们在同步 I/O 支持和锁定上一直都有问题。可以在某些特殊的场合使用它们，比如共享只读的 MyISAM 表。

7.7 使用多个磁盘卷

Using Multiple Disk Volumes

文件放在什么地方的问题早晚都会出现。MySQL 创建了不同的文件：

- 数据和索引文件；
- 事务日志文件；
- 二进制日志文件；
- 普通日志文件（比如错误日志、查询日志、慢速查询日志）；
- 临时文件和表。

MySQL 没有很多管理复杂的表空间的特性。在默认情况下，它简单地把每个数据库（架构）的文件放到单个目录下。有几个选项可以控制数据的位置。比如，可以为 MyISAM 表定义索引位置，也可以使用 MySQL5.1 的分区表。

327

如果使用 InnoDB 的默认配置，所有的数据和索引都会被放到一组文件中，只有表定义文件会被放到数据库目录中。这样的后果就是大部分人都把所有数据和索引放到一个卷里面。

但是，有时使用多个卷可以帮助管理繁重的 I/O 负载。例如，在不同卷上执行一批把数据写入一个巨型表的操作会比较有好处，因为它不会让其他查询的 I/O 操作进入等待状态。在理想的状态下，应该对不同数据的 I/O 访问进行分析，这样就可以把数据放在合适的地方。但是除非已经把数据放在不同的卷上，否则就无法做到这一点。

你也许已经听到过标准的建议，就是把事务日志和数据文件放到不同的卷上，所以日志文件的顺序 I/O 就不会影响对数据进行的随机 I/O。但是如果没有很多硬盘（20 个左右），在这么做之前就要想清楚。

将日志和数据分开真正的优势在于不会在机器崩溃的时候同时丢失日志和数据。如果 RAID 控制器没有电池备用写入缓存，那么它就是很好的实践。但是如果有电池备用单元，分离的卷也许就不是那么需要了。性能通常不是决定性因素，因为尽管事务日志写入数量很大，但它们大多都较小。这样，RAID 缓存就可以把请求合并到一起，通常每秒进行几次顺序物理写入。这不会对数据文件的随机 I/O 造成真正的影响。普通日志通常有顺序的异步写入和低负载，它们也能顺畅地和数据一起共享卷。

但是可以从另一方面来看待这个问题，很多人都没想到这一点。把日志文件放到不同的卷上能提高性能吗？一般说来是可以的，但是因为性能值得这么做吗？回答通常为否。

其原因是为事务日志设置专属硬盘代价较高。假设有 6 个硬盘，明显的想法就是把这 6 个硬盘放到一个 RAID 卷中，或者把其中 4 个留给数据，两个留给事务日志。但是如果这么做，留给数据的硬盘就只有 3 个了，这是很明显的下降；同样地，有两个硬盘被留给了微不足道的负载（假设 RAID 控制器有电池备用的写入缓存）。

从另一方面来说，如果有许多硬盘，那么把其中的一些专门留给事务日志是合适的，也是有益处的。比如，总计有 30 个硬盘，那么把其中两个（配置为 RAID 1 卷）专门留给日志会保证尽可能快的日志写入速度。为了得到额外的性能提升，还应该 RAID 控制器内把一些写入缓存专门留给这个 RAID 卷。

成本效益并不是唯一考虑。另外一个把 InnoDB 数据和事务日志保存在同一个卷上面的原因是它可以为无锁定备份使用 LVM 快照。一些文件系统允许连续的多卷快照，对于这些系统，这不是问题，但是需要注意 ext3 系统。

如果打开了 `sync_binlog`，那么二进制日志在性能方面就和事务日志差不多。但是，把二进制日志保存到和数据不同的卷里实际是一个好主意，这样会更安全，即使数据丢失时，它们也还存在。这种方式可以用于时间点恢复。这种想法不适用于 InnoDB 事务日志，因为它们必须依赖于数据。不能把事物日志使用在昨晚的备份上（对于熟悉其他数据库产品的 DBA 来说，事务日志和二进制日志之间的区别显得很假，因为它们在其他系统中都是一样的。）

分隔文件的唯一其他场景就是临时目录，MySQL 使用它来进行文件排序和保存临时表。如果不是过于巨大，可以把它们放到只使用临时内存的文件系统中，比如 `tmpfs`。这会是更快的选择。如果这在系统中不可行，那就把它放在操作系统所在的设备中。

典型的磁盘架构就是把操作系统、交换分区和二进制日志放到 RAID 1 卷上，并且把其他的所有东西都放在 RAID 5 或 RAID 10 卷上。

7.8 网络配置

Network Configuration

就像延迟和吞吐量是硬盘的限制因素一样，延迟和带宽（和吞吐量是一回事）是网络连接的限制因素。对于大多数程序，延迟是最大的问题。典型的应用程序会进行很多小的网络传输，并且微小的延迟会累加起来。

没有正确地运转的网络会成为主要性能瓶颈。丢包是常见的问题，即使是 1% 的丢包率也会造成明显的性能下降，因为传输协议的不同层次都会尝试修复这个问题，比如进行等待或重新发包，这需要额外的时间。另外一个常见的问题就是网络中断或域名（DNS）解析出错。

DNS 已经足够成为程序的致命弱点，打开 `skip_name_resolve` 对于生产服务器是好主意。失效或缓慢的域名解析对于大多数程序都是问题，但对 MySQL 尤其严重。当 MySQL 收到连接请求的时候，它会进行正向和反向的 DNS 查找。有很多因素会导致出错。当错误发生的时候，连接就会被拒绝，减慢了连接到服务器的过程，并且通常会导致灾难，后果甚至会像受到拒绝服务攻击一样。如果打开 `skip_name_resolve` 选项，MySQL 就不会进行任何 DNS 查找。但是，这也意味着用户帐户的主机栏只能包含 IP 地址、“localhost”或 IP 地址通配符。任何一个主机栏有主机名的用户都不能登录。

需要对网络进行设计来得到好的性能，而不是只是使用默认值。在开始的阶段，要对节点之间的中继的数量进行分析，还要描绘出物理网络的布局。例如，假设有 10 个网页服务器通过千兆网连接到一个网络交换器上，并且该交换器也通过千兆网连接到数据库交换器上。如果不花时间对连接进行跟踪，那么也许永远都意识不到从

所有数据库服务器到所有网页服务器的总带宽只有 1GB！每个中继点也会增加延迟。

在所有网络端口上监视网络性能和错误是一个很好的主意，要监控服务器、路由和集线器上的每一个端口。MRTG（Multi Router Traffic Graper, <http://oss.oetiker.ch/mrtg/>）是对设备进行监控的好工具。另外一些监控网络性能（不是监控设备）的常用工具是 smokeping (<http://oss.oetiker.ch/smokeping/>) 和 cacti (<http://www.cacti.net>)。

网络中物理隔离也有较大的影响。城际网络通常比数据中心的局域网延迟要大得多，尽管两者理论上的带宽是一样的。如果节点在地域上确实离得很远，那么光速也会造成影响。假设在美国西海岸和东海岸都有数据中心，它们之间的距离大约是 4 800 公里。光速是每秒 30 万公里，所以单程至少需要 16 毫秒，来回就是 32 毫秒。地址位置并不是影响性能的唯一因素，两地之间的设备也会影响性能。转发器、路由器和集线器都会在一定程度上降低性能。还有，网络节点之间距离越大，连接就越不可靠和不可预测。

避免跨数据中心的实时数据操作是一个好主意（注 6）。如果非要这么做，那么就要让应用程序能妥善地处理网络错误。例如，你不会想让自己的服务器运行太多的 Apache 进程，因为它们在连接远程数据中心的时候，如果丢包情况严重，它们就会全部挂起。

在本地使用 1GB 带宽的千兆网。对于交换机之间的骨干连接，也许会需要 10GB 带宽的千兆网。如果还需要更大的带宽，就可以使用网络中继，也就是用多网卡实现更大的带宽。中继从本质上说是网络连接并行化，它对实现高可用性很有帮助。

在需要很高的吞吐量的时候，也许可以通过调整操作系统的网络设置改进性能。如果没有很多连接，但是有很大的查询或结果集，那么就可以增加 TCP 缓冲区的大小。具体的更改方式和操作系统有关，但是在大部分 GNU/Linux 系统上，可以更改/etc/sysctl.conf 的值并且执行 sysctl -p，或者利用/proc 文件系统，并且把值传递到 /proc/sys/net/下的文件中。可以用“TCP tuning guide”进行搜索并找到一些好的教程。

但是更重要的通常是调整自己的设置，以高效处理大量的连接和小查询。一个常见的调整方法是改变本地端口的范围。下面是把系统配置到默认值的例子：

```
[root@caw2 ~]# cat /proc/sys/net/ipv4/ip_local_port_range
32768 61000
```

有时也许需要改大一点，比如：

```
[root@caw2 ~]# echo 1024 65535 > /proc/sys/net/ipv4/ip_local_port_range
```

可以按照下面的方式增加队列允许的连接数：

```
[root@caw2 ~]# echo 4096 > /proc/sys/net/ipv4/tcp_max_syn_backlog
```

对于只在本地使用的数据块服务器，可以缩短一方已经断开连接，但是另一方还在等待的超时。在大部分系统上默认值是 1 分钟，这长了一点，可以按照下面的方式修改：

```
[root@caw2 ~]# echo <value> > /proc/sys/net/ipv4/tcp_fin_timeout
```

在大部分时候，这些设置都可以使用默认值。通常只有在某些事情不正常的时候才需要改变它们，比如极差的网络性能或大量的连接。在网上用“TCP variables”进行搜索，就能找到很多很好的阅读材料。

注 6：复制不是跨数据中心的实时数据操作。它不是实时的，并且出于安全考虑，将数据复制到远程服务器是个不错的注意。下一章会讨论该话题。

7.9 选择操作系统

Choosing an Operating System

GNU/Linux 是高性能 MySQL 服务器中最常见的操作系统，但是 MySQL 也可以运行在许多其他的操作系统上。

Solaris 是采用 SPARC 硬件的领先操作系统，它经常被用于高可靠性系统。Solaris 在某些方面比 GNU/Linux 难用一些，但是它是一种拥有很多先进功能的高性能、高可靠性服务器。

331 特别值得一提的是，Solaris 10 很流行。它有自己的文件系统（ZFS）、许多处理故障工具（例如 DTTrace）、良好的线程管理，以及有助于资源管理的，名为 Solaris Zones 的虚拟化技术。Sun 对 MySQL 也提供了很好的支持。

FreeBSD 是另外一个选择。它的旧版本对 MySQL 的支持有很多问题，大部分都和线程支持有关，但是新版本好了很多。现在在 FreeBSD 上大规模地部署 MySQL 已经不常见了。

Windows 在使用 MySQL 进行桌面应用程序开发的时候也很常用。有公司在 Windows 系统上面部署 MySQL 进行企业应用，但没有 Unix 系统那么常见。我们在这儿不是要挑起操作系统之争，在各种操作系统上部署 MySQL 都是可以的。在 Unix 系统和 Windows 系统上运行 MySQL 都非常合理，可以通过 ADO.NET 连接器（它在 MySQL 中是附带的）连接到服务器。从 Unix 机器连接到安装在 Windows 中的 MySQL 服务器和连接另外一台 Unix 服务器一样容易。

在选择操作系统的时候，如果使用了 64 位硬件，那么就应该安装系统的 64 位版本。这听上去很傻，但是我们经常看到在 64 位系统中安装的是 32 位操作系统。处理器的运行通常不会有任何问题，但是 32 位系统的局限（比如可寻址内存大小）会导致 64 位芯片不能发挥最大的作用。

对 GNU/Linux 的各种发布版本，个人偏好通常是决定因素。最佳的选择是使用专门为服务器设置的版本，而不是专门为桌面应用程序发布的版本。也要考虑系统的生命周期、发布和更新策略，同时还要检查供应商的支持情况。Red Hat 的企业版本质量很高，而且很稳定。CentOS 是很流行（同时也免费）的替代系统。Ubuntu 现在正流行。

7.10 选择文件系统

Choosing a Filesystem

文件系统的选择非常依赖于操作系统。在很多系统中，比如 Windows，其实只有一到两种选择。在另一方面，GNU/Linux 支持很多文件系统。

许多人都想知道什么文件系统在 GNU/Linux 上能让 MySQL 发挥最好性能，或者更明确一点，就是想知道什么能让 InnoDB 和 MyISAM 发挥最佳性能。实际测试表明大部分文件系统的性能都非常接近。

332 为了性能而寻找文件系统实际上很不划算。文件系统的性能和工作负载有关，而且没有哪一种文件系统是万用灵丹。在大部分时候，一种文件系统不会比另外的系统性能更好或更差。例外情况就是遇到了文件系统限制，比如系统如何处理并发、处理大量文件、碎片等等。

更重要的是考虑崩溃恢复时间和是否遇到了特定的限制，例如文件夹中有大量文件时性能低下（这在 ext2 和 ext3 文件系统中是很有名的问题，但是现在 ext3 好了一些）。文件系统对于数据安全非常重要，因此我们强烈推荐

不要在产品系统上做实验。

在可能的时候，最好使用报表型文件系统，比如 ext3、ReiserFS、XFS、ZFS 或 JFS。如果不这样，崩溃之后的文件系统检查会花很长的时间。如果系统不是非常重要，那么非日志文件系统的性能可能会好一些。例如，ext2 的性能可能会比 ext3 好，或者可以使用 tune2fs 关闭 ext3 的日志特性。对于某些系统，挂载时间也是一个因素。例如，ReiserFS 会花费较长的挂载时间，并且在分区上面执行日志恢复。

如果使用 ext3，有 3 个选项决定数据如何被记录，可以把它们放在/etc/fstab 挂载选项里面：

Data=writeback

该选项意味着只有元数据写入会被记录。元数据写入和数据写入并不同步。这是最快的选项，并且通常对于使用 InnoDB 是安全的，因为它有自己的事务日志。例外情况就是在某些时候发生的崩溃会损坏.frm 文件。

这儿有一个配置文件如何引发问题的例子。假定某个程序打算扩展一个文件，元数据（文件大小）将会在数据被实际写入新文件之前被记录下来，结果就是文件的尾部包含了垃圾。

Data=ordered

该选项也只会记录元数据，但是它在写入元数据之前写入数据，所以能保证数据的一致性。它比 writeback 选项稍稍慢一点，并且在崩溃的时候要安全得多。

在这个配置中，如果假设程序打算扩展文件，文件的元数据在实际数据被写入新扩展的区域之前是不会改变的。

Data=journal

该选项提供了原子性日志行为，在数据被写入最终位置之前会被写入日志中。通常没有必要使用，它的开销比前两种要高得多。但是，在某些情况下，它可以提高性能，因为它会让文件系统延迟数据写入。

不管是什么文件系统，都有一些特定的选项最好处于关闭状态，因为它们不会带来任何的好处，却会增加开销。最著名的是记录访问时间，这即使是在读取文件的时候也会发生写入。为了关闭这个选项，可以在/etc/fstab 中加入 noatime 选项。这有时会提升 5%~10% 的性能，具体情况取决于工作负载和文件系统（但是，在另外的情况下可能会有很大的差别）。下面是我们提到的 ext3 系统的/etc/fstab 的示例：

```
/dev/sda2 /usr/lib/mysql ext3 noatime,data=writeback 0 1
```

还可以调整文件系统的预读行为，因为它可能是多余的。例如，InnoDB 自己就会进行预读。关闭或限制在 Solaris 的 UFS 上的预读尤其有好处，可以使用 O_DIRECT 自动关闭预读。

一些文件系统不支持所需要的特性。例如，在对 InnoDB 使用 O_DIRECT 刷写方法的时候，直接 I/O 操作可能是很重要的（更多相关内容请参阅第 288 页“InnoDB 如何打开并清写日志和数据文件”）。还有，一些文件系统能更好地处理大量的驱动器。例如，XFS 在这一点上通常好于 ext3。最后，如果计划使用 LVM 快照来初始化从服务器或进行备份，那么就应该选择适合 LVM 版本的文件系统。

表 7-2 总结了一些常用文件系统的特性。

333

表 7-2：常用文件系统特性

| 文件系统 | 操作系统 | 记录日志 | 大型目录 |
|---------------|-----------------|------|-------|
| Ext2 | GNU/Linux | 否 | 否 |
| Ext3 | GNU/Linux | 可选 | 可选/部分 |
| HFSPlus | MacOS | 可选 | 是 |
| JFS | GNU/Linux | 是 | 否 |
| NTFS | Windows | 是 | 是 |
| ReiserFS | GNU/Linux | 是 | 是 |
| UFS (Solaris) | Solaris | 是 | 可调整 |
| UFS (FreeBSD) | FreeBSD | 否 | 可选/部分 |
| UFS2 | FreeBSD | 否 | 可选/部分 |
| XFS | GNU/Linux | 是 | 是 |
| ZFS | Solaris、FreeBSD | 是 | 是 |

334

7.11 线程处理

Threading

MySQL5.0 为每一个连接使用一个线程，再加上内部使用的线程、有特殊目的的线程，以及其他任何由存储引擎创建的线程，MySQL 需要对大量线程进行有效管理。它确实需要对核心级别线程的支持，同时也需要对用户线程的支持，所以它能有效地使用多个 CPU。它也需要有效的同步元素，比如互斥量。操作系统的线程库必须能提供这些。

GNU/Linux 提供了两个线程库：LinuxThreads 和较新的 NPTL (Native Posix Threads Library)。LinuxThreads 在有些情况下还在使用，但是大多数现代版本都转向了 NPTL，而且很多发布版本根本就没有包含 LinuxThreads。NPTL 通常更为轻量级，也更有效率，同时还没有 LinuxThreads 存在的诸多问题。它有一些性能方面的缺陷，但是大多数问题都已经得到了解决。

FreeBSD 也发布了一些线程库。它曾经对线程的支持得很差，但是现在好多了，而且在某些测试中，它甚至在 SMP 系统上超过了 GNU/Linux。在 FreeBSD6 及以上版本中，推荐的线程库是 libthr，早期版本应该使用 linuxthreads，它是 FreeBSD 上的 LinuxThreads。

Solaris 对线程有很好的支持。

7.12 交换

Swapping

当物理内存的数量不能容纳数据的时候，操作系统就会把虚拟内存中的数据写到磁盘上，此时就会发生交换（注 7）。交换对于运行在操作系统中的进程是透明的。只有操作系统知道特定的虚拟内存地址是在物理内存中还是

注 7：交换有时也叫分页。从技术上说来，它们是不同的动作，但是人们通常认为它们是一样的。

磁盘上。

交换对 MySQL 性能有很坏的影响，它使在内存中进行缓存变得毫无意义，而且比对缓存使用了过低内存的性能更低。MySQL 和存储引擎有很多算法来处理内存中和磁盘上的数据，它们都基于内存中的数据访问速度更快这一前提。由于交换对于用户进程是不可见的，MySQL（或存储引擎）不会知道原本应该在内存中数据已经被移到了磁盘上。

这会导致很差的性能。比如，如果存储引擎认为数据还在内存中，它就会认为在进行短期“内存”操作的时候锁住全局互斥量（例如 InnoDB 的缓冲池互斥量）是可行的。如果操作导致了磁盘 I/O，那么所有的动作都必须等到 I/O 完成之后才能进行。这意味着交换比简单地进行 I/O 操作影响更坏。

在 GNU/Linux 上，可以使用 `vmstat` 监视交换（下节有一些示例）。你需要检查 `si` 和 `so` 栏的 I/O 交换活动，而不是 `swpd` 列的交换使用情况。`swpd` 栏能够显示已经被加载却未被使用的进程，那通常没什么问题。`si` 和 `so` 的值最好是 0，而且每秒必须要小于 10 个数据块。

在极端的情况下，太多的交换会耗尽操作系统的交换空间。如果这种情况发生了，它会导致虚拟内存过低，从而导致 MySQL 崩溃。但是即使没有耗尽交换空间，非常活跃的交换也会导致操作系统失去响应，这时候甚至不能登陆系统杀掉 MySQL 进程。

可以通过配置 MySQL 缓冲区解决大部分交换问题，但是有时操作系统的虚拟内存系统会决定对 MySQL 采取交换。这通常发生在操作系统发现 MySQL 进行了大量的 I/O 活动，所以决定增加文件缓存以容纳更多数据。如果没有足够的内存，一些数据就会被交换出去，并且这些数据可能就属于 MySQL。一些老的 Linux 内核版本也有一些不利于性能的交换行为，但是在最近的核心中，这个问题轻多了。

一些人支持完全禁止文件交换。这在某些极端的情况下，系统核心会强制进行交换，这会降低操作系统的性能。（在理论上这不可能，但实际中会发生。）它也很危险，因为禁止交换会给虚拟内存设定不灵活的上限。如果 MySQL 在内存消耗上有临时的尖峰，或者如果在同一台机器上有很多非常需要内存的进程（例如晚间进行的批处理工作），MySQL 可能就会耗尽内存、崩溃，或者被操作系统强制杀掉。

操作系统通常对虚拟内存和 I/O 进行某些控制。我们提到了在 GNU/Linux 系统上的一些控制手段。最基本的就是把 `/prodsys/vm/swappiness` 设置为较小的值，比如 0 或 1。这告诉核心只有在极端需要虚拟内存的时候才进行交换。例如，下面是一个检查当前值并把它设置为其他值的例子：

```
$ cat /proc/sys/vm/swappiness
60
$ echo 0 > /proc/sys/vm/swappiness
```

另外一种方式就是告别存储引擎读取和写入数据的方式。例如，使用 `innodb_flush_method=0_DIRECT` 会减轻 I/O 压力。直接 I/O 不会被缓存，所以操作系统不会把它看成增加文件缓存的原因。这个参数只对 InnoDB 有效，尽管 Falcon 已经支持直接 I/O 了。还可以使用大页面，它们是不可交换的。这对 MyISAM 和 InnoDB 都有效。

另外的选项是使用 MySQL 的配置选项，它会把 MySQL 锁定在内存中。这会避免交换，但有可能是危险的。如果没有足够的可锁定内存，MySQL 在试图分配更多内存的时候就会崩溃。如果太多内存被锁定，留给操作系统的内存不足，也会发生这样的问题。

许多问题都和核心版本有关，所以要小心一些，特别是在升级的时候。在某些负载下，很难让操作系统合理地工作，唯一的办法也许就是把缓冲区的大小降低到一个不是最佳的值。

7.13 操作系统状态

Operating System Status

操作系统可能提供了工具来确定它和硬件正在进行的工作。我们使用示例展示了如何使用两个广为人知的工具：iostat 和 vmstat（注 8）。如果系统没有提供它们，也有可能提供了类似的工具。这儿的目的并不是想使你成为使用 iostat 和 vmstat 的专家，而是向你展示如何使用这样的工具来诊断问题。

除了这些工具之外，操作系统可能还提供了另外的工具，比如 mpstat 或 sar。如果对系统其他部分有兴趣，比如网络，那么就应该使用别的工具，比如 ifconfig（它能显示网络错误的数量，还有其他信息）或 netstat。

在默认情况下，vmstat 和 iostat 只会产生一个报告，显示了从服务器启动以来各种计数器的平均值，这并不是很有用。但是，可以给这两个工具附加时间间隔参数，这样就可以产生增量结果，显示服务器现在正在进行的事情，它和调优更为相关。（第一行显示了自系统启动以来的统计结果，可以简单地忽略它。）

7.13.1 如何读取 vmstat 的输出

How to Read vmstat Output

先看一个 vmstat 的例子。为了每 5 秒钟生成一个新的报告，可以使用下面的命令：

```
$ vmstat 5
procs -----memory----- --swap-- ----io---- -system-- ----cpu----
 r  b   swpd   free   buff   cache   si   so    bi    bo    in   cs us sy id wa
 0  0   2632  25728  23176  740244   0    0   527   521   11    3 10  1 86  3
 0  0   2632  27808  23180  738248   0    0    2   430  222   66  2  0 97  0
```

337 可以使用 Ctrl-C 停止 vmstat。你所看到的输出可能会因为操作系统而有很大差异，所以应该阅读手册以了解详情。

正如前文所述，即使要求增量输出，第 1 行仍然是自从系统启动以来的平均值。第 2 行显示了当前正在发生的事情，并且接下来的行显示了每 5 秒间隔发生的情况，表头各列的含义是：

Procs

栏 r 显示了有多少进程正在等待 CPU 时间。栏 b 显示了处于不可中断的休眠的进程数量，这通常意味着它们在等待 I/O（磁盘、网络、用户输入等）。

Memory

Swpd 栏显示了被交换到磁盘的数据块的数量。剩下的 3 列显示了未被使用的数据块、用于缓冲区的数据块、用于操作系统的数据块的数量。

Swap

这些列显示了交换动作：操作系统每秒从磁盘上交换到内存和从内存交换到磁盘的数据块的数量。监视它

注 8：这儿展示了 vmstat 和 iostat 的结果，因为它们使用非常广泛，并且 vmstat 在很多 Unix 系统上都是默认安装的。但是，每一种工具都有它的局限性，例如让人困惑的度量单位；当操作系统更新统计的时候，在间隔内的抽样不会发生相应的变化，并且不能一次性看到所有的指标。如果这些工具不能满足要求，那么你可能会对 dstat (<http://dag.wieers.com/home-made/dstat>) 或 collectl (<http://collectl.sourceforge.net/>) 感兴趣。

们比监视 `swpd` 列要重要得多。

在大部分时间，`s1` 和 `s0` 最好都是 0，并且每秒肯定不能超过 10 个数据块。突然增加也不好。

Io

这些列显示了每秒从设备中读入（`b1` 栏）和写入到设备（`b0` 栏）的数据块的数量。这通常反映了磁盘 I/O。

System

这些列显示了每秒发生的中断的数量（`in` 列）和上下文交换（`cs` 栏）的数量。

Cpu

这些列显示了用于运行用户（非核心）代码、运行系统（核心）代码、空闲、等待 I/O 的 CPU 时间。可能还有第五列（`st`），如果使用了虚拟化，它会显示从虚拟机“偷来”的百分比。这显示了虚拟机在可以运行某些程序时，却进行了其他操作的时间。如果虚拟机不想运行任何程序并且管理器运行了其他的东西，它就不是“偷来”的时间。

`Vmstat` 的输出依赖于系统，因此如果你的结果和示例中的不一样，那么应该阅读系统的 `vmstat(8)` 参考页。一个重要的提示是内存、交换区、I/O 统计都是以块为单位的，在 Linux 系统上，一块通常是 1024 个字节。

7.13.2 如何读取 `iostat` 输出

How to Read `iostat` Output

现在看看 `iostat`（注 9）。在默认情况下，它和 `vmstat` 一样，显示了同样的 CPU 使用信息。但是我们通常会对 I/O 统计感兴趣，因此可以使用下面的命令只显示扩展的设备统计信息：

```
$ iostat -dx 5
Device: rrqm/s  wrqm/s  r/s  w/s  rsec/s  wsec/s  avgrq-sz  avgqu-sz  await  svctm  %util
sda         1.6     2.8  2.5  1.8  138.8   36.9    40.7     0.1  23.2   6.0   2.6
```

和 `vmstat` 一样，第一行显示了自从系统启动以来的平均统计（这儿已经略去了这一行），接下来的行显示了增量数据。每一个设备是一行。

有不同的选项可以显示或隐藏数据列，上面例子中显示的列为：

Rrqm/s 和 wrqm/s

每秒合并的写入和读取请求的数量。“合并”意味着操作系统接受了多个逻辑请求，并且把它们合并为单个对实际设备的请求。

R/s 和 w/s

每秒发送给设备的读写请求的数量。

Rsec/s 和 wsec/s

每秒发生的扇区读写的数量。一些系统也输出为 `rkb/s` 和 `wkb/s`，指每秒读写的 KB 的数量。这儿也忽略了它们。

注 9：这儿列出的 `iostat` 的统计为了印刷需要进行了一些处理。为了避免折行，小数的位数被减少了。

Avgrq-sz

扇区中的请求大小。

Avgqu-sz

在设备队列中等待的请求数量。

Await

响应请求需要的毫秒数，包括队列时间和服务时间。不幸的是，iostat 没有对读写请求显示各自的服务时间，它们有很大不同，因此不应该用一个平均时间来表示。也许可以把高 I/O 等待看成读取，因为写入通常都是被缓冲过的，但读取通常都是被直接服务的。

Svctm

从头到尾服务请求的时间，包括队列时间和设备实际用来满足请求的时间。

%util

在请求发起期间，CPU 的使用百分比。就像列名一样，这通常显示了设备的利用率，因为当值接近 100% 时，CPU 的使用就达到饱和了。

可以使用输出来推断出机器 I/O 子系统的一些情况。一个重要的指标就是能服务的并发请求的数量。因为每秒进行的读写和服务时间的单位都是每秒千，下面的时间可以抵消单位并且显示设备能服务的并发请求的数量(注 10)：

$$\text{并发} = (r/s + w/s) * (svctm/1000)$$

这儿有一个 iostat 输出的例子：

| Device: | rrqm/s | wrqm/s | r/s | w/s | rsec/s | wsec/s | avgrq-sz | avgqu-sz | await | svctm | %util |
|---------|--------|--------|-----|-----|--------|--------|----------|----------|-------|-------|-------|
| Sda | 105 | 311 | 298 | 820 | 3236 | 9052 | 10 | 127 | 113 | 9 | 96 |

把数字代入上面的公式中，计算出并发数量是 9.6 (注 11)。这意味着平均说来，设备在抽样时间间隔内可以服务 9.6 个并发请求。这个例子来自一个由 10 个磁盘组成的 RAID 10 卷，因此操作系统对请求的并行化处理得很好。从另一方面来说，下面这个设备却是串行的：

| Device: | rrqm/s | wrqm/s | r/s | w/s | rsec/s | wsec/s | avgrq-sz | avgqu-sz | await | svctm | %util |
|---------|--------|--------|-----|-----|--------|--------|----------|----------|-------|-------|-------|
| Sdc | 81 | 0 | 280 | 0 | 3164 | 0 | 11 | 2 | 7 | 3 | 99 |

并发公式计算的结果表明设备每秒只能处理一个请求。这两个设备的利用率基本都达到了饱和，但是它们的性能有很大不同。如果设备在所有的时间都很繁忙，这时就应该检查并发情况，以确定它是否接近于物理设备的数量。值较低则说明有问题。

7.13.3 CPU 密集型机器

• CPU-Bound Machine

对于 CPU 密集型的机器，vmstat 输出的 us 列的值通常较高，它显示了非内核代码执行的时间。在大多数情况下都有一些进程排队，等待 CPU 时间（显示在 r 列）。下面有一个例子：

注 10：另外一种计算并发的方式是通过队列的平均大小、服务时间和平均等待：(avuqu_sz * svctm) / await。

注 11：如果用这些数据实际计算一下，它的值约为 10，这是由于格式的需要，我们对 iostat 的输出进行了圆整，实际的值是 9.6。


```
$ vmstat 5
procs -----memory----- --swap-- -----io----- --system-- -----cpu-----
 r b  swpd free buff cache si so  bi bo  in  cs us sy id wa
10 2  740880 19256 46068 13719952 0 0  2788 11047 1423 14508 89 4 4 3
11 0  740880 19692 46144 13702944 0 0  2907 14073 1504 23045 90 5 2 3
7 1  740880 20460 46264 13683852 0 0  3554 15567 1513 24182 88 5 3 3
10 2  740880 22292 46324 13670396 0 0  2640 16351 1520 17436 88 4 4 3
```

注意到有很多上下文切换（Cs 列）。上下文切换发生在操作系统停止一个进程，然后启动另外一个进程的时候。

如果看看 iostat 在同一台机器上的输出（再次忽略第一行，它显示了自从启动以来的平均值），可以看到磁盘利用率低于 50%。

```
$ iostat -dx 5
Device:rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sda      0      3859   54  458   2063   34546      71      3      6      1      47
dm-0      0      0     54 4316   2063   34532      8     18      4      0      47

Device:rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
sda      0     2898   52  363   1767   26090      67      3      7      1     45
dm-0      0      0     52 3261   1767   26090      8     15      5      0     45
```

这台机器不是 I/O 密集型的，但是它仍然执行了相当数量的 I/O 操作，这对于数据库服务器并不常见。从另一方面来说，典型的网页服务器会消耗大量的 CPU 资源，但是进行很少的 I/O 操作，因此网页服务器的输出通常和这个例子不一样。

7.13.4 I/O 密集型机器

An I/O-Bound Machine

在 I/O 密集型负载中，CPU 花费了大量的时间等待 I/O 请求完成。这意味着 vmstat 将会显示大量的进程处于不可中断的休眠中（列 b），并且 wa 列的值会较高，这儿是一个例子：

```
$ vmstat 5
procs -----memory----- --swap-- -----io----- --system-- -----cpu-----
 r b  swpd free buff cache si so  bi bo  in  cs us sy id wa
5 7  740632 22684 43212 13466436 0 0  6738 17222 1738 16648 19 3 15 63
5 7  740632 22748 43396 13465436 0 0  6150 17025 1731 16713 18 4 21 58
1 8  740632 22380 43416 13464192 0 0  4582 21820 1693 15211 16 4 24 56
5 6  740632 22116 43512 13463484 0 0  5955 21158 1732 16187 17 4 23 56
```

这台机器的 iostat 输出表明磁盘已经达到饱和了：

```
$ iostat -dx 5
Device:rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
Sda      0     5396  202  626   7319   48187      66     12     14      1    101
dm-0      0      0    202 6016   7319   48130      8     57      9      0    101

Device:rrqm/s wrqm/s r/s w/s rsec/s wsec/s avgrq-sz avgqu-sz await svctm %util
Sda      0     5810  184  665   6441   51825      68     11     13      1    102
dm-0      0      0    183 6477   6441   51817      8     54      7      0    102
```

%util 值由于圆整错误可能会超过 100%。

对于 I/O 密集型的机器，这个结果意味着什么？如果有足够的缓冲区容量为写入请求服务，它通常——但却不是总是——意味着磁盘不能满足读取请求，即使机器正在执行大量的写入操作。除非理解了读写的本质，否则就会觉得这是违背直觉的：

- 写入请求能被缓冲或同步。它们可以在本书其他地方讨论的任何一个层次被缓存起来，比如操作系统、RAID 控制器等。
- 读取请求本质上是同步的。有可能一个程序预测到会需要一些数据，并且进行了预提取（Prefetch）。但是，更常见的情况是程序在继续工作之前才发现需要数据。这就强制把请求变成了同步的：程序只有在请求完成的时候才不会被阻塞。

可以这样思考这个问题：你发起了写入请求，它被写入缓冲区中，稍后就会被完成。每秒可以发起很多这样的请求。如果缓冲区工作正常并且有足够的空间，每个请求都能很快被完成，并且出于性能考虑，这些请求会被重新排序，然后成批地写入物理磁盘。

但是，没办法对读取做同样的事情，不管它们数量是多少，也不管它们是多小，都没办法进行缓冲。这就是为什么读取会造成大部分 I/O 等待的原因。

7.13.5 交换型机器

A Swapping Machine

正在执行交换的机器可能会，也可能不会在 `swpd` 列显示较高的值。但是，在 `si` 和 `so` 列的值会较高，它不是你想要的。下面是一个进行繁重交换的机器的 `vmstat` 输出：

```
$ vmstat 5
procs -----memory----- --swap-- ----io---- --system-- ----cpu----
 r  b  swpd free buff cache  si  so   bi   bo   in   cs us sy id wa
 0 10 3794292 24436 27076 14412764 19853 9781 57874 9833 4084 8339 6 14 58 22
 4 11 3797936 21268 27068 14519324 15913 30870 40513 30924 3600 7191 6 11 36 47
 0 37 3847364 20764 27112 14547112 171 38815 22358 39146 2417 4640 6 8 9 77
```

7.13.6 空闲机器

An Idle Machine

出于完整性考虑，下面有一个处于空闲的机器的 `vmstat` 输出。要注意没有不可运行的或被阻塞的进程，并且 `idle` 列显示 CPU 处于 100% 空闲。这个例子来自于一台运行 Red Hat Enterprise Linux 5 的机器，它显示了 `st` 列，那表示了从虚拟机“偷来”的时间：

342

```
$ vmstat 5
procs -----memory----- --swap-- ----io---- --system-- ----cpu----
 r  b  swpd   free  buff  cache  si  so   bi   bo   in   cs us sy id wa st
 0  0  108  492556  6768 360092   0   0   345  209   2   65  2  0  97  1  0
 0  0  108  492556  6772 360088   0   0   0   14  357  19  0  0 100  0  0
 0  0  108  492556  6776 360084   0   0   0    6  355  16  0  0 100  0  0
```

MySQL 内建的复制能力是构建大型、高性能应用程序的基础。复制 (Replication) 使你可以为一台服务器配置一台或多台从服务器 (Slave Server)。这不仅仅对高性能的程序有益, 对其他任务也很方便, 比如和远程办公室共享数据、保持“热备份 (Hot Spare)”, 或者在服务器上保留一份用于测试或培训的数据备份。

本章阐述了复制的方方面面。首先是复制工作方式的综述, 接下来介绍了服务器的基本配置, 设定更高级的复制配置选项, 以及管理和优化复制服务器。本书大部分内容都集中在性能上, 但是我们同样重视复制的正确性和可靠性, 因此展示了让复制更好地工作的建议。我们还研究了和 MySQL 复制有关的一些即将实现的改变和改进, 比如 Google 创建的一些有趣的补丁。

8.1 复制概述

Replication Overview

复制解决的基本问题是让一台服务器的数据和另外的服务器保持同步。一台主服务器可以连接多台从服务器, 并且从服务器也可以反过来作主服务器。主服务器和从服务器可以位于不同的网络拓扑中, 还能对整台服务器、特定的数据库, 甚至特定的表进行复制。

MySQL 支持两种复制方案: 基于语句复制 (Statement-Based Replication) 和基于行复制 (Row-Based Replication)。基于语句复制在 MySQL 3.23 就已经存在, 它也是现在使用得最多的复制方式。基于行复制是 MySQL 5.1 的新特性。这两种方式都是通过记录主服务器的二进制日志 (注 1), 并在从服务器上进行重放 (Replay) 完成复制, 它们都是异步的。这就是说, 从服务器上的数据并非随时都是最新的 (注 2), 同样也不能保证从服务器上的延迟具体大小。大型查询能让从服务器落后主服务器几秒、几分, 甚至几小时。

344

MySQL 复制大部分都是后向兼容的。这意味着版本较新的服务器可以是版本较老的服务器的从服务器。但老版本的服务器通常不能是新版本服务器的从服务器, 原因是它可能不理解新服务器使用的新特性或语法, 而且它们之间使用的复制文件的格式还可能不一样。例如, 你不能从 MySQL 5.0 复制到 MySQL 4.0。在进行版本升级前, 比如从 4.1 升级到 5.0, 或者从 5.0 升级到 5.1, 最好能对复制进行一些测试。

复制通常不会大幅增加主服务器的开销。它需要主服务器启用二进制日志, 这或许会有较大的开销, 但是出于备份目的, 无论如何这种开销都是需要的。除了二进制日志, 每个连接的从服务器都会在主服务器的正常操作期间增加一点负载 (大部分都是网络 I/O)。

复制对扩展读取有好处, 可以把读取指向到从服务器上, 但是只有进行了正确的设计, 复制才是扩展写入的好

注 1: 如果不了解二进制日志, 可以查阅第 6 章、本章其余部分, 以及第 11 章得到更多信息。

注 2: 更多内容请参看第 451 页的“MySQL 同步复制”。

方法。把许多从服务器附加到主服务器只是简单地导致了多次写入，每个从服务器一次。整个系统的性能取决于这些写入中最慢的一环。

复制在从服务器较多的时候也比较浪费，因为它本质上是复制了大量不需要的数据。例如，一台主服务器有 10 台从服务器，它有 11 份相同数据的拷贝，并且在 11 个不同的缓存中重复了相同的数据。这和服务器端有 11 路 RAID 1 类似。这对使用硬件并不经济，但是它在复制的设置中非常常见。本章我们讨论了解决这个问题的方法。

8.1.1 复制解决的问题

Problems Solved by Replication

下面是复制通常的用途：

数据分布 (Data Distribution)

MySQL 的复制通常不会对带宽造成很大的压力（注 3），并且可以随意启动和停止。因此，它对于在不同的地方维护数据拷贝很有作用，比如在各个数据中心之间，远方的从服务器甚至可以使用断断续续的连接。但是，如果需要从服务器有很低的复制延迟，那么就需要稳定、低延迟的连接。

345 负载均衡 (Load Balancing)

MySQL 复制可以把读取分布在不同的服务器上，这对读取密集型的程序效果很好。可以对代码做一些简单的更改实现基本的负载均衡。可以使用小规模의 更改，比如在代码里面硬编码主机名并使用循环域名服务 (Round-Robin DNS)（它把一个主机名指定给多个 IP 地址）。还可以采用复杂的方式，例如实现负载均衡的网络产品也能在 MySQL 服务器之间分布负载。Linux 虚拟服务器 (Linux Virtual Server LVS) 也能很好地工作。第 9 章详细讨论了负载均衡。

备份 (Backups)

复制对备份很有帮助。但是从服务器既不是备份，也不是备份的替代品。

高可用性和故障转移 (High Availability and Failover)

复制可以避免在应用程序中出现 MySQL 失效。好的故障转移系统包括了能显著减少故障停机时间的复制服务器。第 9 章也讨论了故障转移。

测试 MySQL 升级 (Testing MySQL Upgrade)

一个常见的实践就是在把所有的服务器升级到新版本之前，使用从服务器安装 MySQL 升级版本，然后用它来测试查询，确保查询按照想要的方式来工作。

8.1.2 复制如何工作

How Replication Works

在进入详细设置复制之前，先看看它实际是如何复制数据的。总体上说来，复制有 3 个步骤：

注 3：但是 MySQL 5.1 中引入的基于行复制会比传统的基于命令复制需要更多的带宽。

1. 主服务器把数据更改记录到二进制日志中。（这叫做二进制日志事件（Binary Log Events）。）
2. 从服务器把主服务器的二进制日志事件拷贝到自己的中继日志（Relay Log）中。
3. 从服务器重放中继日志中的事件，把更改应用到自己的数据上。

这只是概述，每一个步骤都很复杂。图 8-1 更详细地描述了复制。

第 1 步是在主服务器上记录二进制日志（稍后说明如何设置它）。在每个更新数据的事务完成之前，主服务器都会把数据更改记录到二进制日志中。即使事务在执行期间是交错的，MySQL 也会串行地把事务写入到二进制日志中。在把事件写入二进制日志之后，主服务器告诉存储引擎提交事务。

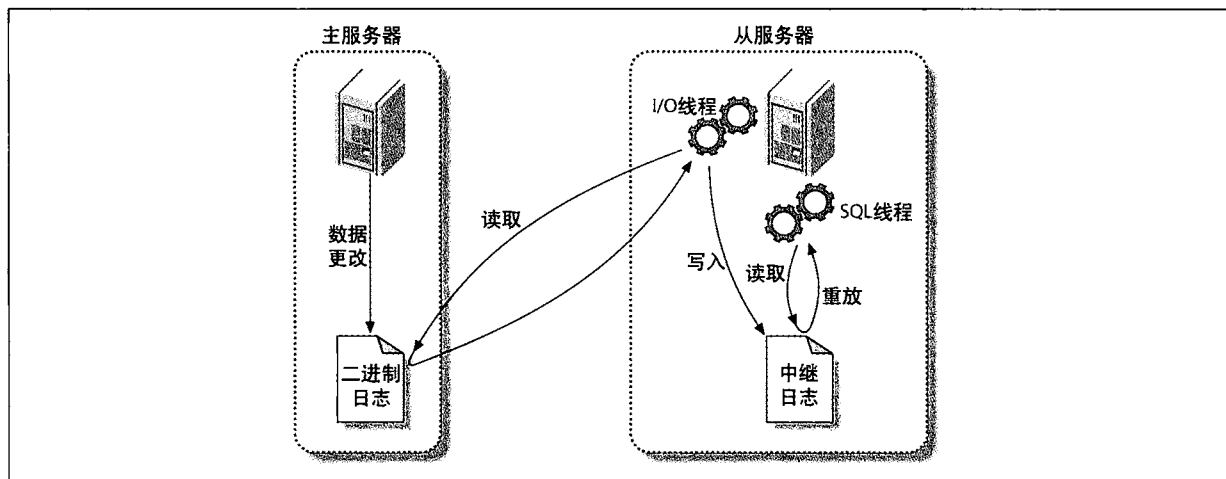
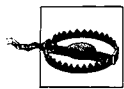


图 8-1: MySQL 复制工作方式

下一步是从服务器把主服务器的二进制日志拷贝到自己的硬盘上，进入所谓的“中继日志（Relay Log）”中。首先，它启动一个工作线程，叫 I/O 从线程（I/O Slave Thread）。这个 I/O 线程开启一个普通的客户端连接，然后启动一个特殊的二进制日志转储（Binlog Dump）进程（它没有相应的 SQL 命令）。这个转储进程从主服务器的二进制日志中读取事件。它不会对事件进行轮询。如果它跟上了主服务器，就会进入休眠状态，并等待有新事件发生时主服务器发出的信号。I/O 线程把事件写入从服务器的中继日志中。



警告：MySQL 4.0 之前的复制在很多方面都不一样。例如，MySQL 第 1 个复制功能没有使用中继日志，因此复制只使用了两个线程，而不是 3 个。大部分人使用的都是新版本的服务器，所以我们不会在本章谈及老版本的更多细节。

SQL 从线程处理了该过程的最后一部分。该线程读取了中继日志，并且重放其中的事件，然后更新从服务器的数据。由于这个线程能跟上 I/O 线程，中继日志通常都在操作系统的缓存中，所以中继日志的开销很低。SQL 线程执行的事件也可以被写入从服务器自己的二进制日志中，它对于我们稍后讲到的场景很有用。

图 8-1 只显示了从服务器上运行的两个复制线程，但是在主服务器上还有一个线程。和连接到 MySQL 的任何连接一样，从服务器打开的连接也会在主服务器上启动一个线程。

这种复制架构在从服务器上实现了提取和重放事件之间的解耦，这允许它们之间是异步的。也就是说，I/O 线程可以独立于 SQL 线程之外工作。它也对复制进程加以了限制，最重要的是从服务器上的复制是串行的。这意味着主服务器上并行运行（在不同的线程中）的更新在从服务器上不能并行进行。稍后我们会看到，这对于很多工作负载都是瓶颈。

8.2 创建复制

Setting Up Replication

在 MySQL 中创建复制是很简单的，但是由于场景不同，基本的步骤还是有很多差异。最基本的场景是刚刚安装好的主服务器和从服务器。总的说来，分为下面几步：

1. 在每一台服务器上建立复制账号。
2. 配置主、从服务器。
3. 指导从服务器进行连接与复制。

这儿假设了许多默认设置都是足够的，如果主、从服务器刚刚安装好，并且它们有相同的数据（默认的 mysql 数据库），那么这种假设是对的。我们先假设服务器是 server1（IP 地址：192.168.0.1）和 server2（IP 地址：192.168.0.2）。接下来我们解释了如何从一台已经在运行的服务器中初始化从服务器，并且探寻了推荐的复制配置。

8.2.1 创建复制账号

Creating Replication Accounts

MySQL 有一些特殊的权限允许复制进程运行。从 I/O 线程运行在从服务器上，它创建了到主服务器的连接。这意味着必须在主服务器上创建一个用户账户并且给它合适的权限，这样 I/O 线程就可以以这个用户的身份连接到主服务器，并且读取它的二进制日志。下面是创建用户的例子，用户名是 repl：

```
mysql> GRANT REPLICATION SLAVE, REPLICATION CLIENT ON *.*  
-> TO repl@'192.168.0.%' IDENTIFIED BY 'p4ssword';
```

该命令在主、从服务器上都创建了这个账号。注意到我们把这个用户限制到了本地网络，因为复制账号是不安全的（更多关于安全的内容请参考第 12 章）。

348



提示：复制用户在主服务器上实际只需要 REPLICATION CLIENT 权限，主、从服务器上并不真正需要 REPLICATION SLAVE 权限。那么为什么我们要把这些权限都授予主、从服务器呢？有两个原因：

- 用于监视和管理复制的账号将需要 REPLICATION SLAVE 权限，并且对这两个目的使用同一个账号更容易（而不是为这个目的再开一个单独的账号）。
- 如果在主服务器上建立了账号，然后克隆到从服务器上，从服务器也可以成为主服务器，这样就可以交换主、从服务器的角色。

8.2.2 配置主、从服务器

Configuring the Master and Slave

下一步是在主服务器上开启一些设置，假设主服务器是 server1。这时需要打开二进制日志并定义一个服务器 ID（Server ID）。在主服务器的 my.conf 文件中输入（或者验证）下面的配置行：

```
log_bin      = mysql-bin  
server_id    = 10
```

实际的值取决于你自己。我们这儿只是为了简单起见，你可以定义更精确的值。

必须显式地定义唯一的服务器 ID。我们使用 10 来代替 1，因为 1 通常是服务器的默认值，（这和版本有关，有的 MySQL 版本根本就不能使用 1。）使用 1 很容易引起混淆，并导致和没有服务器 ID 的服务器冲突。通常的办法是使用服务器 IP 地址的最后 8 位二进制数，但是要假设它不会改变并且是唯一的（例如，服务器都在同一个子网里面）。

如果二进制日志选项在主服务器的配置文件中没有定义，那就要重启 MySQL。为了验证二进制日志文件已经被创建出来，可以运行 SHOW MASTER STATUS 命令，并检查输出是否和下面的类似（MySQL 将会在文件名后面添加几个数字，所以你不会看到和自己定义的完全一样的名字）：

```
mysql> SHOW MASTER STATUS;
+-----+-----+-----+-----+
| File           | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+-----+-----+-----+-----+
| mysql-bin.000001 |      98 |              |                  |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

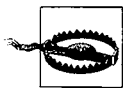
从服务器也需要和主服务器类似的配置，并且也需要在从服务器上重启 MySQL。

```
log_bin          = mysql-bin
server_id        = 2
relay_log        = mysql-relay-bin
log_slave_updates = 1
read_only        = 1
```

从技术上说来，某些选项是不需要的，我们只是显式地列出了默认值。实际上，从服务器只有 server_id 参数是必需的，但是我们也开启了 log_bin，并且显式地给了二进制文件一个名字。默认情况下它和服务器的主机名一样，但是如果主机名变了，它就会出问题。同时，我们把主、从服务器的文件名设置成一样的，以便于把从服务器变为主服务器。因此，除了在主、从服务器上使用一样的账号之外，我们还对主、从服务器使用同样的设置。

我们还添加了两个其他的可选配置参数：relay_log（定义中继日志的位置和文件名）和 log_slave_updates（使从服务器把复制的事件记录到自己的二进制日志中）。后一个选项会导致额外的工作，但是你马上就会看到，我们有理由在每一个从服务器上添加这个可选的参数。

一些人只开启了二进制日志，但是关闭了 log_slave_updates，因此他们可以看到是否有任何东西——比如错误配置的应用程序——正在更改从服务器上的数据。如果可能，最好使用 read_only 配置设置，它会阻止任何没有特殊权限的线程更改数据（不要给用户授予超出他们需要的权限）。但是，read_only 通常不够实际，特别是对于需要在从服务器上创建表的应用程序。



警告：不要把 mast_host 和 master_port 这些复制参数放到从服务器的 my.conf 文件中。这种方式已经被废弃了。它只会导致问题，没有任何好处。

8.2.3 启动复制

Starting the Slave

下一个步骤是告诉从服务器如何连接到主服务器并中继其二进制日志。不要为这个使用 my.conf 文件，而应该使用 CHANGE MASTER TO 命令。这个命令完全代替了相应的 my.conf 设置。它可以让你在以后把从服务器指向

不同的主服务器，并且不用停止服务器。下面是一个开始复制的基本命令：

```
mysql> CHANGE MASTER TO MASTER_HOST='server1',
-> MASTER_USER='repl',
-> MASTER_PASSWORD='p4ssword',
-> MASTER_LOG_FILE='mysql-bin.000001',
-> MASTER_LOG_POS=0;
```

350 MASTER_LOG_POS 参数被设置为 0，因为它在日志的开头。在运行这个命令之后，就可以用 SHOW SLAVE STATUS 检测输出，并且查看从服务器的设置是否正确：

```
mysql> SHOW SLAVE STATUS\G
***** 1. row *****
Slave_IO_State:
  Master_Host: server1
  Master_User: repl
  Master_Port: 3306
  Connect_Retry: 60
  Master_Log_File: mysql-bin.000001
  Read_Master_Log_Pos: 4
  Relay_Log_File: mysql-relay-bin.000001
  Relay_Log_Pos: 4
  Relay_Master_Log_File: mysql-bin.000001
  Slave_IO_Running: No
  Slave_SQL_Running: No
  ...omitted...
Seconds_Behind_Master: NULL
```

Slave_IO_State、Slave_IO_Running 和 Slave_SQL_Running 列显示从服务器进程没有运行。聪明的读者也会注意到日志的位置是 4，而不是 0。这是因为 0 其实不是真正的日志位置，它只是意味着“在日志文件的开头。”MySQL 知道第一个事件的真正位置是 4（注 4）。

运行下面的命令开始复制：

```
mysql> START SLAVE;
```

这个命令应该不会出错，也不会有输出。现在再次使用 SHOW SLAVE STATUS：

```
mysql> SHOW SLAVE STATUS\G
***** 1. row *****
Slave_IO_State: Waiting for master to send event
  Master_Host: server1
  Master_User: repl
  Master_Port: 3306
  Connect_Retry: 60
  Master_Log_File: mysql-bin.000001
  Read_Master_Log_Pos: 164
  Relay_Log_File: mysql-relay-bin.000001
  Relay_Log_Pos: 164
  Relay_Master_Log_File: mysql-bin.000001
  Slave_IO_Running: Yes
  Slave_SQL_Running: Yes
  ...omitted...
Seconds_Behind_Master: 0
```

注 4：实际上，就像在 SHOW MASTER STATUS 中看到的那样，真正的位置是 98。一旦从服务器连接到主服务器，它们将会一起开始工作，这时连接还未发生。

注意到从服务器 I/O 线程和 SQL 线程正在运行，并且 Seconds_Behind_Master 不再是 NULL（稍后再考察 Seconds_Behind_Master 的含义）。I/O 线程正在等待主服务器的事件，这意味着它已经提取了主服务器的所有二进制日志。日志位置已经增加了，表示一些事件已经被提取并执行过了（你的结果可能会有所不同）。如果在主服务器上做一些更改，你应该会看见从服务器上的文件和位置参数都增加了。此时应该可以在从服务器上看到数据库改变！

还可以在主、从服务器的进程列表中看见复制线程。在主服务器上，应该能看见从服务器 I/O 线程创建的连接：

```
mysql> SHOW PROCESSLIST\G
***** 1. row *****
  Id: 55
  User: repl
  Host: slavel.Webcluster_1:54813
  db: NULL
  Command: Binlog Dump
  Time: 610237
  State: Has sent all binlog to slave; waiting for binlog to be updated
  Info: NULL
```

在从服务器上，应该可以看见两个线程。一个是 I/O 线程，另外一个为 SQL 线程：

```
mysql> SHOW PROCESSLIST\G
***** 1. row *****
  Id: 1
  User: system user
  Host:
  db: NULL
  Command: Connect
  Time: 611116
  State: Waiting for master to send event
  Info: NULL
***** 2. row *****
  Id: 2
  User: system user
  Host:
  db: NULL
  Command: Connect
  Time: 33
  State: Has read all relay log; waiting for the slave I/O thread to update it
  Info: NULL
```

这儿显示的实例来自于运行了很久的服务器，这是 I/O 线程的 Time 列值较大的原因。从服务器的 SQL 线程已经空闲了 33 秒，这表示已经有 33 秒没有发生事件重放了。

这些进程总是运行在“system user”账户下，但是其余列的值可能会不同。例如，当 SQL 线程正在重放事件的时候，Info 列将会显示正在执行的查询。



提示：如果只想试验一下 MySQL 复制，Giuseppe Maxia 的 MySQL Sandbox 脚本 (<http://sourceforge.net/projects/mysql-sandbox>) 能很快地从一个下载的 MySQL 包中进行一次性的安装。只需要几次按键和大约 15 秒钟就能得到一个正在运行的主服务器和两个从服务器。命令是：

```
$ ./set_replication.pl ~/mysql-5.0.45-linux-x86_64-glibc23. tar.gz
```

8.2.4 从另一个服务器开始复制

Initializing a Slave from Another Server

前面的设置教程都假设主、从服务器刚刚安装好，使用的数据都是默认的初始化数据，因此两台服务器隐式地有相同数据，并且你也知道主服务器的二进制日志。这不是典型的案例。通常情况是主服务器已经运行了一段时间，然后用一台新安装的从服务器进行同步，此时它还没有主服务器的数据。

有几种方式从另外一台服务器初始化，或者“克隆”一台从服务器。这包括从主服务器拷贝数据，从另外一台从服务器进行克隆，以及从最近的备份中恢复。为了让从服务器和主服务器同步数据，需要 3 个条件：

- 某个时间点上主服务器数据的快照。
- 主服务器当前的日志，以及从快照时间点以来精确的日志字节偏移量。我们把这两个值叫做日志文件坐标 (Log File Coordinates)，因为它们一起确定了二进制日志的位置。可以使用 `SHOW MASTER STATUS` 查找主服务器的日志文件坐标。
- 从快照时间到现在的主服务器二进制日志文件。

下面有一些方式从另外一台服务器克隆一台从服务器：

使用冷拷贝 (With a Cold Copy)

最基本的方式就是关闭主服务器并且把文件拷贝到从服务器（高效拷贝文件的方法请参看附录 A），然后再次启动主服务器，它会启动另外一份新的二进制日志。这次利用 `CHANGE MASTER TO` 在二进制日志的开头启动从服务器。这种方法的劣势很明显：需要在拷贝的时候关闭主服务器。

使用热拷贝 (With a Warm Copy)

如果仅仅用了 MyISAM 表，那么可以使用 `mysqldump` 在服务器正在运行的时候拷贝文件。更多细节请参阅第 11 章。

使用 `mysqldump` (Using `Mysqldump`)

如果只使用了 InnoDB 表，那么可以使用下面的命令来转储主服务器的所有内容，把它装入从服务器中，并且把从服务器的坐标改成主服务器二进制日志的相应位置：

```
$ mysqldump --single-transaction --all-databases --master-data=1  
--host=server1 | mysql --host=server2
```

`--single-transaction` 选项使转储位于事务的开头，可以读取数据。这个选项有可能适合其他的事务存储引擎，但是我们没有测试过。如果没有使用事务表，可以使用 `--lock-all-tables` 选择所有表的一致性转储。

使用 LVM 快照或备份 (With a LVM Snapshot or Backup)

既然知道了相应的二进制日志坐标，就可以使用主服务器的快照或备份来初始化从服务器（如果使用备份，这种方式要求有自从备份以来的所有主服务器的二进制日志）。只需要把备份或快照恢复到从服务器，然后在 `CHANGE MASTER TO` 中使用适当的二进制日志坐标。第 11 章有更多细节。

第 11 章中也讨论了 InnoDB 热备份，它是在只有 InnoDB 表的情况下初始化从服务器的好方法。

使用另外的从服务器 (From Another Slave)

可以使用快照或上面的任意一种技巧从另外的从服务器进行克隆，但是如果使用 `mysqldump`，`--master-data` 选项就不会工作。

此外，不能使用 `SHOW MASTER STATUS TO` 来得到主服务器的二进制日志坐标，而应该用 `SHOW SLAVE STATUS` 在进行快照的时候找到从服务器在主服务器上执行的位置。

克隆的最大缺点就是如果从服务器的数据没有和主服务器同步，就会得到坏的数据。



警告：不要使用 `LOAD DATA FROM MASTER` 或 `LOAD TABLE FROM MASTER`！它们过时、缓慢，并且非常危险，而且它们只适用于 MyISAM。

不管选择了何种技巧，都要习惯它，用文档对它进行记录，或者为它制作脚本。有可能不止一次地执行它，并且在某些东西出错的时候，也要能执行它。

8.2.5 推荐的复制配置

Recommended Replication Configuration

复制有很多参数，并且大部分都会对数据安全和性能有一些影响。稍后我们再解释何种规则会在何时失效。

本节推荐了一种“安全”的复制配置，它能使出问题的几率最小。

354

在主服务器上最重要的二进制日志设置是 `sync_binlog`：

```
sync_binlog = 1
```

这使 MySQL 在每次提交事务的时候把二进制日志的内容同步到磁盘上，所以即使服务器崩溃也会把事件写入日志中。如果关闭这个选项，服务器会少做一些工作，但是二进制日志在服务器崩溃后可能就会被破坏或丢失信息。在不需要成为主服务器的从服务器上，这个选项带来了不必要的开销。它只适用于二进制日志，而不是中继日志。

如果不想在服务器崩溃后表被破坏，我们推荐使用 InnoDB。在表无关紧要的时候，MyISAM 是可以接受的，但是 MyISAM 表在从服务器崩溃后可能处于不连续的状态。一种可能的情况是命令没有完全地应用到一个表或多个表上，那么即使是修复了表，数据也有可能是不连续的。

如果使用 InnoDB，我们强烈推荐在主服务器上使用下面的选项：

```
innodb_flush_logs_at_trx_commit=1 # Flush every log write
innodb_support_xa=1                # MySQL 5.0 and newer only
innodb_safe_binlog                  # MySQL 4.1 only, roughly equivalent to
                                     # innodb_support_xa
```

这是 MySQL 5.0 的默认设置。在从服务器上，我们推荐开启下面的配置选项：

```
skip_slave_start
read_only
```

`Skip_slave_start` 选项会阻止从服务器在崩溃后自动启动，它可以给你机会修复服务器。如果从服务器在崩溃后自动启动，并且处在不一致的状态，它可能就会导致更多的损坏，那么你就不得不把所有的数据都丢掉，然后重新开始。即使打开了建议的所有选项，从服务器还是有可能在崩溃后轻易地损坏，因为中继日志和 `master.info` 不能承受崩溃。它们有可能不会被写到磁盘上，并且没有配置选项可以控制这种行为。（稍后讨论的 Google 补丁解决了这个问题。）

`Read_only` 选项防止大部分用户更改非临时表，例外情况是 SQL 线程和有 SUPER 权限的线程，这是避免给予不同用户 SUPER 权限的一个原因（更多权限相关的内容请参看第 12 章）。

如果从服务器落后主服务器很多，从 I/O 线程就会生成许多中继日志。从服务器的 SQL 线程将会在完成中继之后尽快地删除它们（可以使用 `relay_log_purge` 选项改变这种行为），但是如果它落后太多，I/O 线程可能就会写满磁盘。解决这个问题的办法是 `relay_log_space_limit`。如果所有中继日志的大小超过了该变量规定的值，I/O 线程就会停下来，并且等待 SQL 线程释放一些磁盘空间。

尽管这听上去不错，但它有可能成为隐藏的问题。如果从服务器没有从主服务器提取到所有的中继日志，并且主服务器崩溃的话，所有的日志就都会丢失。除非担心磁盘空间不够，否则最好让中继日志使用它需要的磁盘空间。这也是我们没有把 `relay_log_space_limit` 包含在推荐设置里面的原因。

8.3 揭示复制的真相

Replication Under the Hood

现在关于复制的一些基本概念已经清楚了，接下来我们要深入地了解它。我们要看看它是如何真正地工作的，了解它的优点和缺点，并且考察更多的高级复制配置选项。

8.3.1 基于命令复制

Statement-Based Replication

MySQL 5.0 和早期版本只支持基于命令复制（也叫逻辑复制）。这在数据库世界里并不常见。基于命令复制通过记录主服务器上改变数据的查询来完成工作。当从服务器从中继日志中读取事件并执行它的时候，实际上是执行了和主服务器同样的查询。这种安排既有好处，也有坏处。

最明显的好处就是实现起来非常简单。在理论上，简单地记录并重放改变了数据的命令，会让从服务器和主服务器同步。另外一个好处就是二进制日志里面的事件比较紧凑。因此，相对说来，基于命令复制不会使用太多带宽——更改了上 GB 数据的查询在日志里面可能只有几十个字节长。同样，贯穿本章的 `mysqlbinlog` 工具，是使用基于命令记录的最佳工具。

但是在实际情况中，基于命令复制并不像看上去那么简单，因为主服务器上的许多更改都依赖于查询文本之外的因素。例如，命令在主从服务器上执行的时间可能会稍微或非常不同。这样，MySQL 二进制日志格式包含的就不仅仅是查询文本了，它还会传送一些元数据，比如当前的时间戳。尽管如此，还是有些命令不能被 MySQL 正确地复制，比如使用了 `CURRENT_USER()` 函数的查询。存储过程和触发器在基于命令复制下也有一些问题。

基于命令复制的另外一个问题就是修改必须是串行的。这需要大量的特殊代码、配置设置和额外的服务器特性，包括 InnoDB 的下键锁（Next-key Lock）和自增锁定。并非所有的存储引擎都支持基于命令复制，尽管那些存储引擎是包括 MySQL 5.1 在内的 MySQL 官方发布版本中的。

可以在 MySQL 手册中找到基于命令复制缺点的完整列表。

8.3.2 基于行复制

Row-Based Replication

MySQL 5.1 增加了对基于行复制的支持，它把实际的数据更改记录到二进制日志中，和大部分其他的数据库产品实现复制的方式很相似。这种复制方式自身也有优点和缺点。它最大的好处就是 MySQL 可以正确地复制每

个命令，并且一些命令可以被更有效地复制。主要缺点就是二进制日志会变得很大，并且对更新数据命令的可见度会降低，因此不能使用 `mysqlbinlog` 检查二进制日志。



提示：基于行记录没有后向兼容性。随 MySQL 5.1 发布的 `mysqlbinlog` 工具能够读取以基于行复制的格式记录的事件（它对人是不可识别的，但是 MySQL 服务器可以解释）。但是，早期的 `mysqlbinlog` 不能识别这种格式，在遇到它们的时候，就会抛出错误并退出。

MySQL 能使用基于行复制更高效地复制数据，因为从服务器不用重放更改了主服务器的数据查询。重放一些查询的代价是很高的。例如，下面有一个查询从很大的表中汇总数据到一个较小的表：

```
mysql> INSERT INTO summary_table(col1, col2, sum_col3)
      -> SELECT col1, col2, sum(col3)
      -> FROM enormous_table
      -> GROUP BY col1, col2;
```

想象一下，如果 `enormous_table` 中的 `col1` 和 `col2` 只有 3 种组合，这个查询就会扫描很多行，但是最终只会产生 3 行输出。复制该事件将导致从服务器重复所有的动作，却只产生了几行数据，但是基于行复制的方式，从服务器的开销就会少很多。在这种情况下，基于行复制效率高得多。

在另一方面，下面的事件使用基于命令复制代价会小很多：

```
mysql> UPDATE enormous_table SET col1 = 0;
```

使用基于行复制的代价会很高，因为它会改变每一行：每一行都会被写入二进制日志中，使二进制日志里面记录的事件非常大。这会给记录事件和复制增加很多负担，并且记录速度变慢会减少并发。

因为没有哪一种格式适合所有情况，MySQL 5.1 会在基于命令复制和基于行复制之间动态切换。在默认情况下，它使用基于命令复制，但是当它探测到事件不能使用命令正确地复制的时候，它就会切换到基于行复制。可以通过设置 `binlog_format` 变量控制复制的格式。

对于有基于行的事件的二进制日志，很难进行时间点恢复。但这并不是不可能的。日志服务器对此会有帮助，稍后将讲解更多内容。

在理论上，基于行复制解决了我们稍后会提到的一些问题，但是在实际情况中，我们所知的大部分把 MySQL 5.1 用于生产环境的人还是在使用基于命令复制。因此，现在对基于行复制下结论还为时过早。

8.3.3 复制文件

Replication Files

让我们看看复制使用的文件。我们已经知道二进制日志和中继日志，但还有其他的文件。MySQL 放置它们的地方依赖于你的设置。不同的 MySQL 版本在默认情况下把它们放在不同的目录中，可以在数据文件夹或在包含服务器的 `.pid` 文件的目录（在 Unix 系统上可能是 `/var/run/mysqld/`）中找到它们。它们是：

Mysql-bin.index

一个开启了二进制日志的服务器有一个文件和二进制日志文件同名，但是后缀是 `.index`。该文件记录了磁盘上的二进制日志文件。这儿的 `index` 并不是指表的索引，而是说这个文件中每一行包含了二进制日志文件的文件名。

你可能会认为这个文件是多余的，可以被删掉（毕竟 MySQL 可以在磁盘上找到需要的文件），但不行。MySQL 依赖于这个 index 文件，除非它有记录，否则 MySQL 就识别不了二进制日志文件。

Mysql-relay-bin.index

这个文件是中继日志的索引文件，其作用和二进制日志文件的索引文件一样。

Master.info

这个文件包含了从服务器连接主服务器需要的信息。它的格式是纯文本（每行一个值），并且随 MySQL 版本变化而变化。不要删除它，否则从服务器就不知道在重启后如何连接主服务器。这个文件以纯文本格式记录了复制用户的密码，因此应该对访问加以限制。

Relay-log.info

该文件包含了从服务器的当前二进制日志和中继日志的坐标（例如，从服务器在主服务器上的位置）。同样不要删除它，否则从服务器在重启之后就会忘记复制的位置，并有可能重复复制。

这些文件是记录 MySQL 的复制和日志状态的粗略方式。不幸的是，它们不是写入同步的，所以如果服务器失去了电力，并且文件没有被刷写到磁盘上，服务器重启的时候它们就可能不太精确。

在默认情况下，二进制日志是主机名加上数字后缀，但是一个好办法是显式地在 my.cnf 中加以定义，就像下面这个例子一样：

```
log_bin          # Don't do this, or files will be named after the hostname
log_bin = mysql-bin # This is safe
```

采用这种方式的原因是服务器的主机名改变了就有可能造成问题。我们也建议不要以主机名命名日志文件，换句话说，就是不要显式地定义默认值。相反地，应该选择一个名字，并且广泛地使用它。这会使在机器之间做文件迁移和实现自动化故障转移更智能。

还应该给中继日志（默认情况下，它同样也是以主机名命名的）及相应的.index 文件显式地命名。下面是我们建议的 my.cnf 设置：

```
log_bin          = mysql-bin
log_bin_index    = mysql-bin.index
relay_log        = mysql-relay-bin
relay_log_index  = mysql-relay-bin.index
```

.index 文件实际继承了日志文件的名字，但是显式地给它们取名不会对它们造成伤害。

.index 文件也和其他设置有交互，expire_logs_days 定义了 MySQL 清除过期日志的方式。如果 mysql-bin.index 文件中的文件在磁盘上不存在，自动化清理就不会起作用。事实上，PURGE MASTER LOGS 这时也不会起作用。这个问题的解决办法通常是使用 MySQL 服务器管理二进制日志，这样就不会导致误解。

你需要显式地执行一些日志清理策略，比如 expire_logs_days 或其他的方式，否则 MySQL 二进制日志就会占满磁盘。在执行这个动作的时候要考虑备份的方法。更多关于二进制日志的内容请参阅第 487 页“二进制日志格式”。

8.3.4 发送复制事件到其他服务器

Log_slave_updates 选项可以把一台从服务器变成主服务器。它指导 MySQL 把自己执行的事件写到二进制日

志中，然后自己的从服务器就可以取得这些事件并执行它。图 8-2 显示了这一过程：

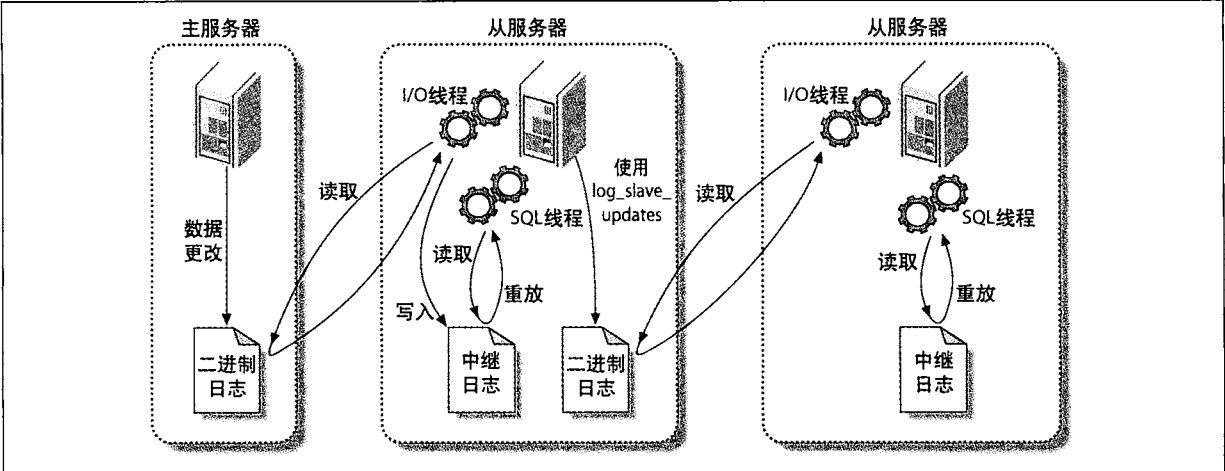


图 8-2：将复制事件传递到更多的从服务器

在这种场景中，主服务器上的改变导致事件被写入到二进制日志中。第 1 个从服务器提取并执行这个事件。通常在这个时候，事件的生命周期就结束了，但是因为 `log_slave_updates` 被打开了，从服务器就会把它写入自己的二进制日志中。现在第 2 个从服务器就可以再次取得这个事件并执行它。这个配置意味着原始的主服务器上的事件可以传播到没有直接连接在它上面的子服务器中。在默认情况下，我们会设置这个选项，这样在连接从服务器后就可以不重启服务器。

在第 1 个从服务器把事件写入自己的二进制日志中时，这个事件的位置几乎肯定和主服务器上的位置不一样，它们有可能在不同的文件中，或者在文件中的不同位置上。这意味着不能假设在同一个逻辑复制点的服务器会有相同的坐标。稍后我们会看到，这使某些任务非常复杂，例如把从服务器变成不同的主服务器，或者把从服务器变成主服务器。

除非已经给每个服务器指定了唯一的服务器 ID，否则按照这种方式配置从服务器会导致微妙的错误，甚至还能使复制停止。一个更常见的问题就是为什么需要定义服务器 ID。MySQL 难道不能在不知道复制命令来源的情况下执行它吗？为什么 MySQL 要在意服务器 ID 是全局唯一的？问题的答案在于 MySQL 在复制中如何防止无限循环。当从 SQL 线程读取中继日志的时候，它会把和自己服务器 ID 匹配的事件丢掉。这在复制的时候打破了无限循环。防止无限循环在某些网络拓扑中尤为重要，例如主—主复制。



提示：如果在建立复制的时候有问题，服务器 ID 就是应该检查的因素之一。只检测 `@@server_id` 变量是不够的。它有默认值。除非在 `my.conf` 中，或者通过 `SET` 命令显式地设置了它，复制才会工作。如果使用 `SET` 命令，要确保也更新了配置文件，否则在服务器重启后设置就会丢失。

8.3.5 复制过滤器

复制过滤选项使你可以只复制一部分数据。有两种过滤器：主服务器上把事件从二进制日志中过滤掉的过滤器和从服务器上把事件从中继日志中过滤掉的过滤器。图 8-3 显示这两种类型。

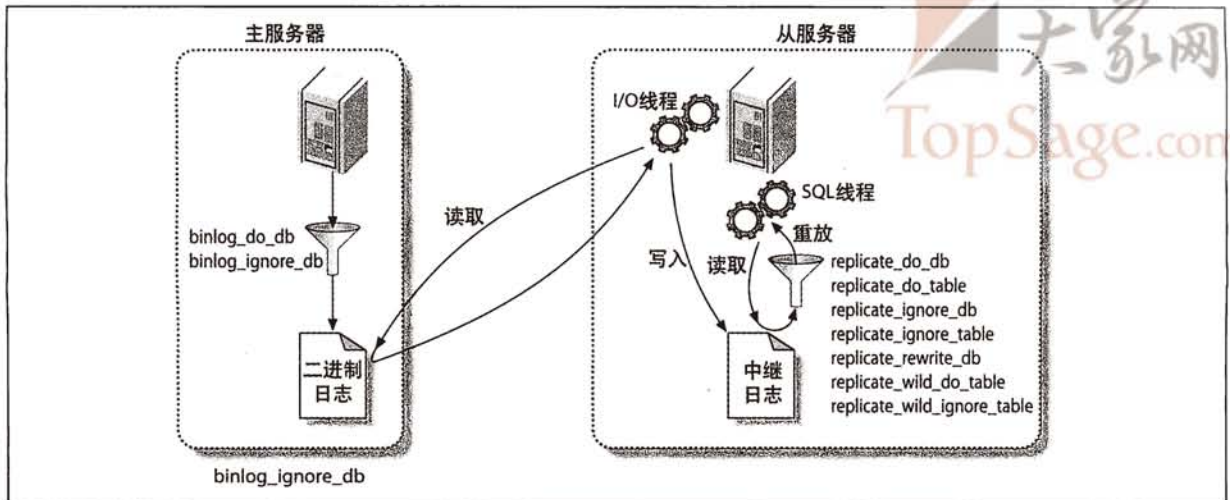


图 8-3: 复制过滤选项

控制二进制日志过滤的选项是 `binlog_do_db` 和 `binlog_ignore_db`。稍后我们会解释为什么通常不需要开启它们。

在从服务器上, `replicate_*` 选项在从服务器 SQL 线程从中继日志中读取事件的时候进行过滤。可以复制, 或者忽略一个或多个数据库, 把一个数据库重写到另一个数据库, 并且使用类似于 LIKE 的模式复制或忽略数据表。

理解这些选项的一个最重要内容就是搞明白 `*_do_db` 和 `*_ignore_db` 的意义, 它们在主、从服务器上都有, 但是有可能不会按照你设想的方式工作。你也许会认为它们会过滤对象的数据库, 但它们实际上过滤的是当前的默认数据库。这就是说, 如果在主服务器上执行下面的命令:

```
mysql> USE test;
mysql> DELETE FROM sakila.film;
```

`*_do_db` 和 `*_ignore_db` 参数将会在 TEST 上过滤 DELETE 命令, 而不是在 sakila 上。这通常不是你想要的结果, 它会导致执行或忽略错误的命令。`*_do_db` 和 `*_ignore_db` 有作用, 但是作用有限, 使用它们的时候要非常小心。如果使用了这些参数, 很容易让复制出错。



警告: `binlog_do_db` 和 `binlog_ignore_db` 选项不会破坏复制, 它们使从备份中按时间点复制变得不可能。在大多数情况下, 都应该不使用它们。本章稍后部分, 我们展示了使用黑洞列表来过滤复制的安全方式。

复制过滤器的一个常用用途是阻止在复制中使用 GRANT 和 REVOKE 命令 (注 5)。通常的问题是管理员在主服务器上给某个用户授予了写入的权限, 但是随后发现这个权限被传递到了从服务器上, 但是此时是不允许修改任何数据的。下面的复制选项会防止这种情况发生:

```
replicate_ignore_table=mysql.columns_priv
replicate_ignore_table=mysql.db
replicate_ignore_table=mysql.host
replicate_ignore_table=mysql.procs_priv
replicate_ignore_table=mysql.tables_priv
replicate_ignore_table=mysql.user
```

注 5: 在从服务器上限制权限的一种较好的方式是使用 `read_only`, 并且在主、从服务器上保持同样的权限。

你可能会看到使用下面的语句可简单地过滤掉 mysql 数据库的所有表：

```
replicate_wild_ignore_table=mysql.%
```

这肯定会在复制过程中阻止 GRANT 命令，但是它也会阻止事件和函数。这些不可预测的因素是我们建议你谨慎使用过滤器的理由。防止某些特定的命令被复制的一种较好的方式通常是使用 SET SQL_LOG_BIN=0，但是这种办法也有自己的缺点。通常说来，应该非常仔细地地使用复制过滤器，而且只在自己真正需要它的时候使用，因为它能轻易地破坏基于命令的复制。（基于行复制也许可以解决一部分这些问题，但是这并没有被完全证明过。）

过滤选项在 MySQL 手册中描述得很仔细，因此这儿不再重复细节。

8.4 复制拓扑

Replication Topologies

基本上可以为任何一种主从服务器配置建立 MySQL 复制机制，局限就是一个给定的 MySQL 从服务器只能有一个主服务器。可能有很多复杂的拓扑结构，但是即使最简单的结构也可能非常灵活。一种拓扑结构也能有很多不同的用途。在阅读本节的时候要记住我们只描述了最简单的用途。使用复制的不同方式可以很轻易地写一本书。

我们已经了解了如何建立一个只有一个从服务器的主服务器。在本节中，我们考察了其他常见的拓扑结构并且讨论了它们的优点和局限。记住下面的基本原则：

- 一个 MySQL 从服务器只能有一个主服务器。
- 每个从服务器有唯一的服务器 ID。
- 一个主服务器可以有很多从服务器（或者说，一个从服务器可以有很多兄弟服务器）。
- 如果打开了 log_slave_updates，一个从服务器就能把主服务器的改动传播下去，并且能成为其他从服务器的主服务器。

8.4.1 主服务器和多个从服务器

Master and Multiple Slaves

这种方式不同于前文描述的主从双服务器配置，它是最简单的复制拓扑结构。事实上，一主多从的结构和基本配置差不多简单，因为从服务器之间根本没有交互。它们只连接到主服务器。图 8-4 显示了这种结构。

这种配置在写入较少，读取较多的时候是最有用的。可以把读取分摊到任意多从服务器上，直到从服务器给主服务器造成了太大的负担，或者主从服务器之间的带宽成为问题为止。可以使用前面的方法一次设立很多从服务器，或者按照需要添加从服务器。

尽管这是非常简单的拓扑结构，但是它足够灵活，可以满足多种需要。下面有一些用途：

- 为不同的角色使用不同的从服务器（例如，添加不同的索引或使用不同的存储引擎）。

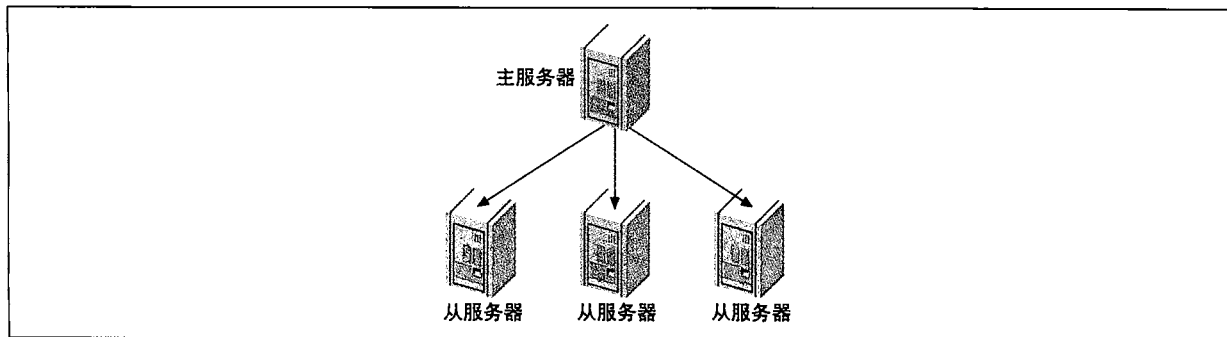


图 8-4：一台主服务器和多台从服务器

- 把一个从服务器当成待命主服务器，除了复制就没有其他的数据流量了。
- 把远程的数据中心作为从服务器，以备灾难恢复。
- 延迟一个或多个从服务器，以备灾难恢复。
- 为备份、培训、开发或测试使用从服务器。

这种拓扑结构流行的原因是它避免了其他拓扑方式带来的复杂性。下面是一个例子：可以方便地在从服务器之间比较二进制日志在主服务器上的位置，因为它们都是完全一样的。换句话说，如果在复制的同一个逻辑点停止所有的从服务器，它们读取的都是主服务器日志的同一个物理位置。这是一个很好的特性，减轻了管理员的很多工作，例如把从服务器变成主服务器。

这种特性只存在于兄弟从服务器之间。在没有直接位于主从服务器关系之内的服务器，或者没有兄弟关系的服务器之间比较日志文件的位置要复杂得多，比如在树形复制或分布式主服务器上了解复制的逻辑顺序就困难得多。

8.4.2 主动—主动模式下的主—主复制

Master-Master in Active-Active Mode

主—主复制（也叫双主服务器复制或双向复制）包含了两个服务器，每一个服务器都既是主服务器，又是对方的从服务器。换言之，它们是一对主服务器。图 8-5 显示了它的结构。

主动—主动模式下的主—主复制有它的用途，但是它们通常都有特殊的用途。一个可能的用途就是两个在不同地方的办公室，每个办公室都需要自己本地的可写的数据拷贝。

这种配置的最大问题是如何解决冲突。由两个可写的主服务器引发的问题列表会很长。

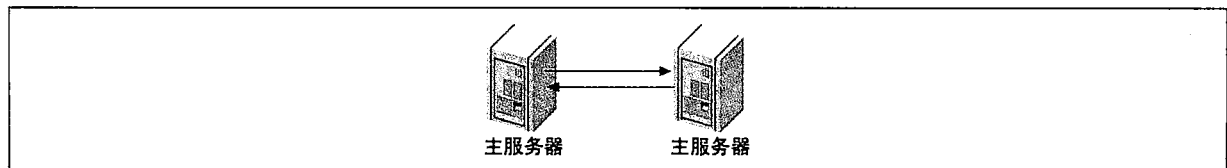


图 8-5：主—主复制

问题通常发生在两个服务器同时更改了同一行数据，或者同时向一个有 `AUTO_INCREMENT` 列的表中插入了数据。

MySQL 不支持多主服务器复制

多主服务器复制 (Multimaster replication) 特指一个从服务器有多个主服务器。不管别人怎么说, MySQL (和其他数据库产品不一样) 现在不支持图 8-6 中的结构。但是, 本章稍后部分会介绍一些方法来模拟多主服务器复制。

不幸的是, 许多人只要是主服务器多于一个, 就使用了这个术语, 例如本章稍后会介绍的树形拓扑结构。还有人把这儿的主—主服务器复制也当成了这种方式。

该术语引起了很多困惑甚至争论, 因此我们最好谨慎地使用这个名字。只要想想如果 MySQL 增加了对一个从服务器加两个主服务器的支持, 沟通将会多么困难。如果不把这个术语保留起来, 那么该用什么术语来描述这种情况呢?

MySQL 增加了一些复制特性, 使这种类型的复制稍微安全了一点。它们是 `Auto_increment_increment` 和 `auto_increment_offset`。这两个选项使服务器对于 `INSERT` 查询可以自动生成不会冲突的值。但是, 允许同时写入仍然是危险的。在两个服务器上按照不同的顺序进行的更新也能导致数据不同步。例如, 假设有个表只有一行和一列, 并且值为 1。现在假设下面的两个命令同时执行:

- 在第 1 个主服务器上:
`mysql> UPDATE tbl SET col=col + 1;`
- 在第 2 个主服务器上:
`mysql> UPDATE tbl SET col=col * 2;`

结果是什么? 一个服务器的值为 4, 而另外一个为 3。并且这时根本就没有复制错误。

数据不同步仅仅是一个开始。如果复制因为错误而停止, 但是应用程序还在向两个服务器写入数据, 那会发生什么情况? 不能简单地进行拷贝, 因为每个服务器上都有需要拷贝到对方机器上的数据。解决这个问题可能会很困难。

如果想很仔细地搭建这种配置, 可能可以通过分区数据和权限避免一些问题 (注 6)。但是, 它很难做好, 并且有更好的方式实现你的目标。

通常说来, 运行在两台机器上写入数据带来的麻烦会多过它的好处。但是, 下一节描述的主动—被动模式会非常有用。

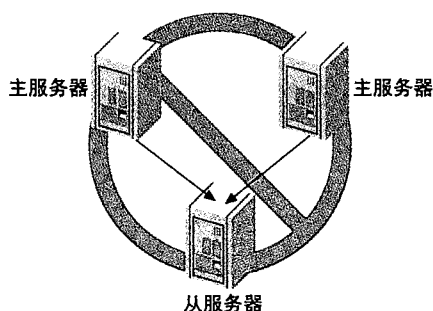


图 8-6: MySQL 不支持多主服务器复制

注 6: 一些, 但不是全部——我们可以吹毛求疵, 并指出任何一种设置的漏洞。

8.4.3 主动—被动模式下的主—主复制

MySQL Master-Master Replication

它是对前面的主—主复制的一种变体，事实上，它是设置容错和高可用性系统的非常强大的方式。主要的区别是一个服务器是只读的“被动”服务器，就像图 8-7 所示。

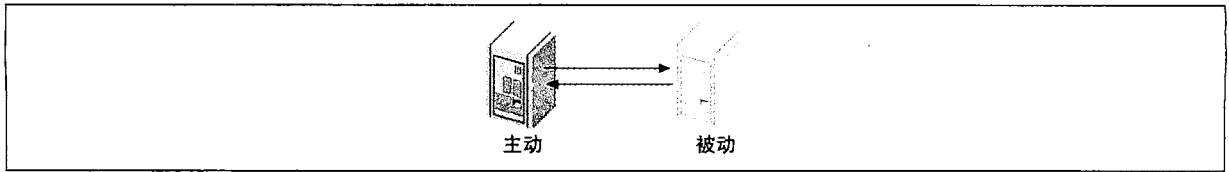


图 8-7：主动—被动模式下的主—主复制

这种配置可以轻易地来回交换主动服务器和被动服务器的角色，因为服务器的配置是对称的。这时故障转移和故障恢复变得容易了。它也让你可以在不关闭服务器的情况下执行维护、优化表、升级操作系统（或者应用程序，硬件）及其他任务。

366

例如，运行 `ALTER TABLE` 命令锁住整个表，阻塞了读写。这可能会花很长时间并中断服务，但是主—主配置可以让你在主动服务器上停止从服务器线程，所以被动服务器就不会处理任何更新操作，在被动服务器上改变表，交换彼此的角色，然后在先前的主动服务器上启动从服务器进程（注 7）。现在该服务器就可以读取中继日志并且执行同样的 `ALTER TABLE` 命令。这同样可能会花很长时间，但是没有关系，因为服务器没有为任何活动的查询提供服务。

主动—被动模式的主—主拓扑可以绕过 MySQL 的许多其他问题和局限。可以使用 MySQL 主—主复制管理工具 (<http://code.google.com/p/mysql-master-master>) 来创建和管理这种拓扑结构。它使很多复杂的动作变得自动化了，例如恢复和重新同步复制、创建新的从服务器等。

让我们看看如何配置主—主服务器对。对两台服务器都执行下面的操作，这样它们就会拥有对称配置：

1. 启用二进制日志、选择唯一的服务器 ID、添加复制账号。
2. 启用记录从服务器更新功能。这对故障转移和故障恢复很关键。
3. 可选性地把被动服务器设置为只读，以防止和主动服务器产生冲突。
4. 确保服务器有相同的数据。
5. 启动服务器上的 MySQL。
6. 将每个服务器设置为对方的从服务器，使用新创建的二进制日志工作。

现在可以追踪一下主动服务器发生更改时会发生什么事情。更改被记录到二进制日志中，并且通过复制传递到了被动服务器的中继日志。被动服务器执行查询并把事件写到自己的二进制日志中，这是因为启用了 `log_save_updates`。主动服务器然后通过复制取得了事件，并写入自己的中继日志中，但是它忽略了这些事件，因为服务器 ID 和自己的一样。

如何切换角色请参阅第 382 页“改变主服务器”。

注 7：也可以通过 `SET SQL_LOG_BIN = 0` 暂时禁止二进制日志，不用停止复制。一些命令，比如 `OPTIMIZE TABLE` 也支持 `LOCAL` 或 `NO_WRITE_TO_BINLOG` 这些停止日志的选项。

创建主动—被动模式下的主—主拓扑在某种程度上有点像创建热备份，但是可以使用这个“备份”来提高性能。可以把它用于读取查询、备份、“离线”维护、升级等，但真正的主—主热备份做不到这些事情。然而，对于写入查询，它的性能不会高于单个服务器（稍后详细解释）。

在讨论使用复制的场景和用途的时候，我们会再次回到这种结构上面。它非常重要并且非常常见。

8.4.4 有从服务器的主—主复制

Master-Master with Slaves

一种相关的配置是给主服务器对添加从服务器，如图 8-8 所示。

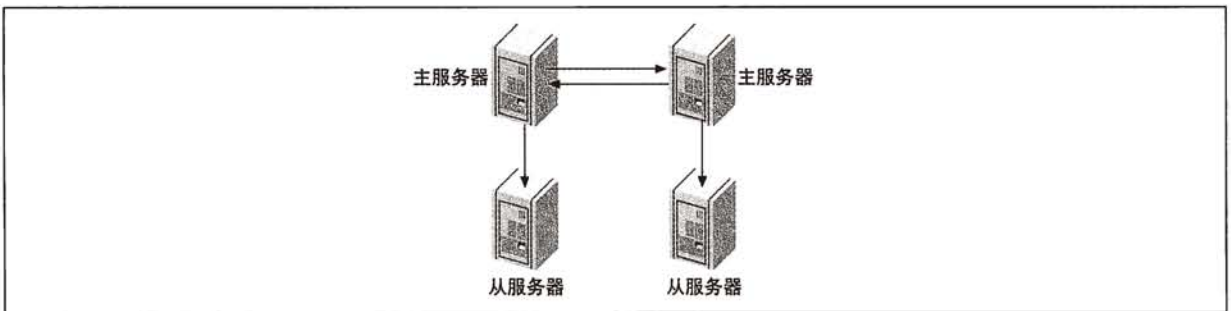


图 8-8：有从服务器的主—主复制

这种配置的好处就是额外的冗余度。对于位于不同地点的复制拓扑，它在每个站点都能消除一个故障点。也可以像平常那样把复制分摊到从服务器上。

如果在本地为了故障转移使用主—主复制，这种方式还是有用的。用其中一个从服务器来代替失效的主服务器是可能的，只是有点复杂。同样也可以把其中一个从服务器变成不同的主服务器，主要的顾虑是它增加的额外复杂度。

8.4.5 环

Ring

环形结构如图 8-9 所示。双主服务器结构其实是环形结构的一种特例（注 8）。环形结构有 3 个或更多的主服务器。每个服务器都是它前面服务器的从服务器，也是后一个服务器的主服务器。这种拓扑也叫环形复制（Circular Replication）。

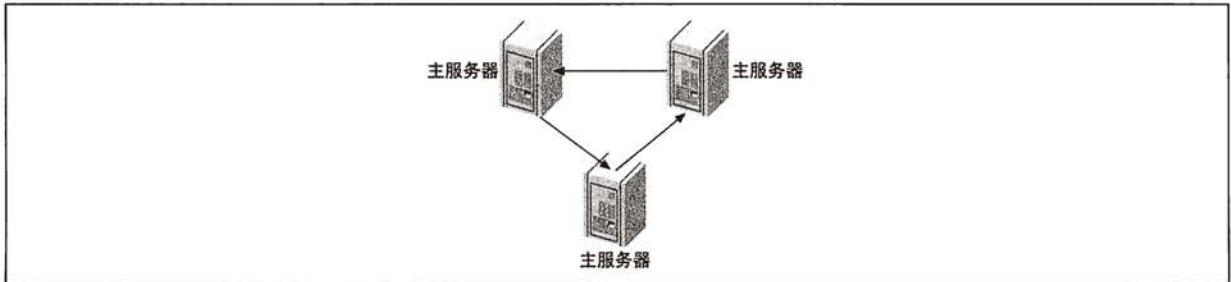


图 8-9：复制环拓扑

注 8：我们也许会说，是更为明智的特例。

环形结构没有主—主结构的某些好处，比如对称和易于故障转移。它也完全依赖于环上的每一个可用的节点，这大大增加了整个系统失效的几率。如果把环上的某一个节点移除，这个节点发起的事件就会进入无限循环，原因是唯一可以根据服务器 ID 过滤掉事件的是服务器，也就是节点自身。通常说来，最好避免环形结构。

可以通过给每一个主服务器添加从服务器以提供冗余，从而减轻一些风险。结构如图 8-10 所示。但是这仅仅防范了服务器失效的风险。停电或其他影响站点连接的问题还是会破坏整个环。

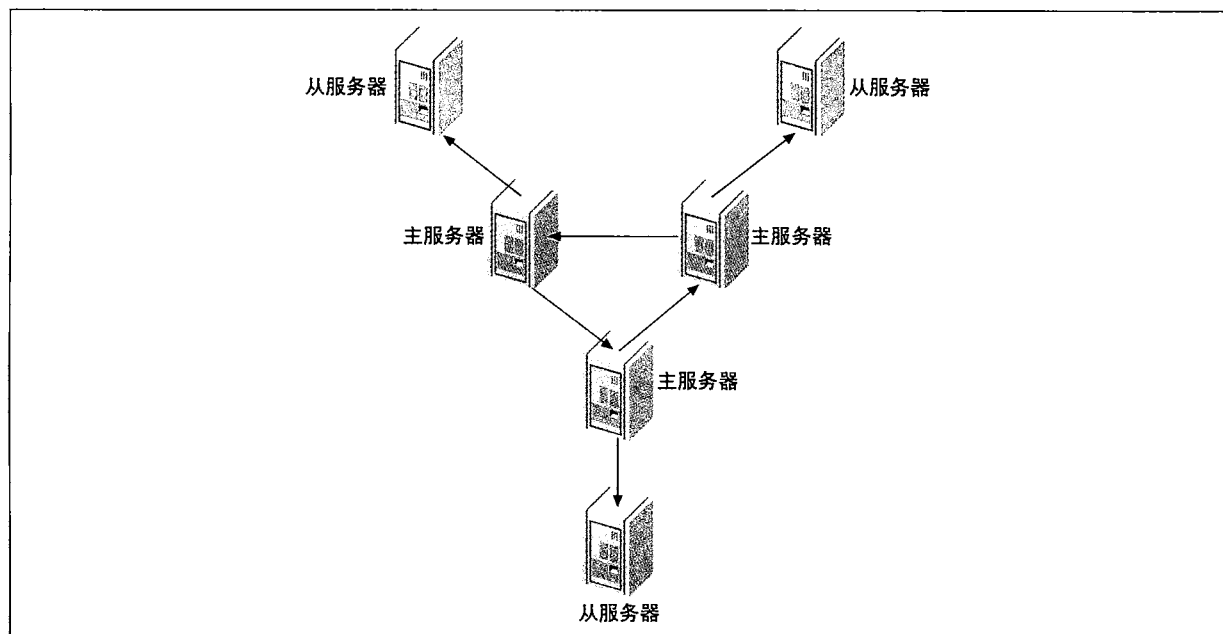


图 8-10: 拥有从服务器的环形结构

369 8.4.6 主服务器、分发主服务器和从服务器

我们已经提到过，如果从服务器的数量够多，它们就会给主服务器带来很大的负担。每个从服务器都会在主服务器上创建新的线程，它会执行特殊的 `binlog dump` 命令。该命令从二进制日志中读取数据并且发送给从服务器。每个从服务器线程都会重复这个工作，它们不会彼此共享资源。

如果有很多从服务器并且有特别巨大的二进制日志事件，例如巨大的 `LOAD DATA INFILE`，主服务器的负担会显著上升。主服务器也许会因为所有从服务器同时请求同样的事件而耗尽内存，甚至崩溃。另外，如果从服务器请求不在文件系统缓存中的不同事件，那么就会导致大量的磁盘搜索，它也会干扰主服务器的性能。

出于这种原因，如果需要很多从服务器，那么一个好主意就是把负载从主服务器上移走，单独添加一个分发主服务器（Distribution Master）。分发主服务器是从服务器，它的唯一目的就是读取和提供主服务器的二进制日志。大量从服务器可以连接到分发主服务器，它使原来的主服务器摆脱了巨大的负担。为了从分发主服务器上移除实际执行查询的功能，应该让它使用黑洞（Blackhole）存储引擎，如图 8-11 所示。

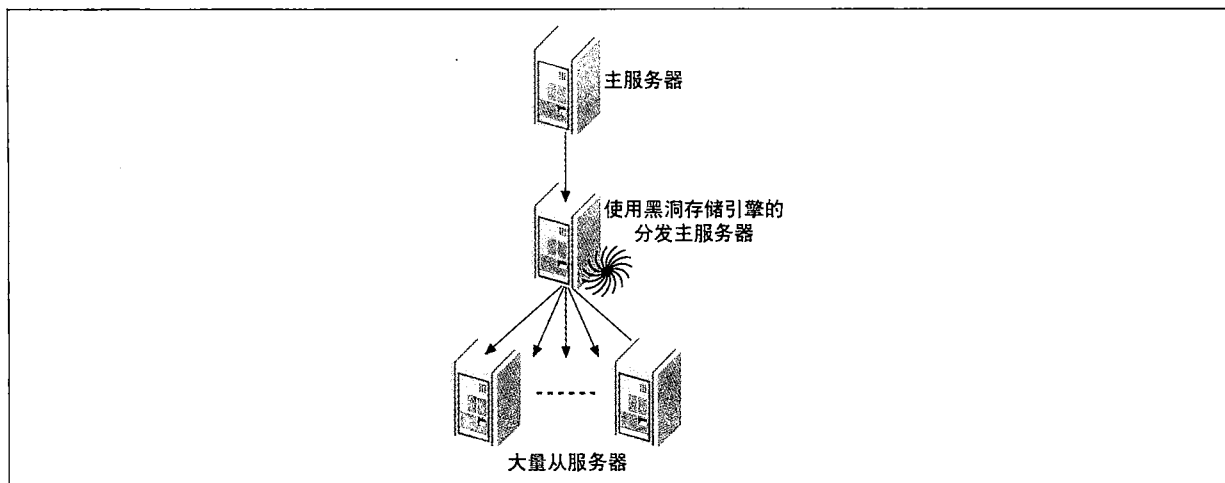


图 8-11：一台主服务器、一台分发主服务器，以及大量的从服务器

很难说有多少从服务器时就会需要分发主服务器。按照通用原则，如果主服务器接近于满载，那么从服务器的数量可能就不能多于 10 个。如果写入活动很少，或者仅仅复制一部分表，主服务器也许就可以支持更多从服务器。另外，也不必只用一台分布主服务器。如果对大量的从服务器做复制，就可以使用几台分布主服务器，或者使用金字塔型的分布主服务器。

也可以为其他目的使用分布主服务器，例如为二进制日志事件应用过滤和重写规则。这比在每台从服务器上重复进行日志记录、重写和过滤要高效得多。

如果在分布主服务器上使用黑洞表（Blackhole Table），它就能比未使用的时候支持更多的从服务器。分布主服务器会执行查询，但是查询的代价极低，因为黑洞表没有任何数据。

一个常见的问题就是如何在使用了黑洞存储引擎的分布主服务器上保护所有的表。如果有人在主服务器上创建了一个新表，并定义了不同的存储引擎，情况将会怎样？确实，不管什么时候在从服务器上使用不同的存储引擎都会导致同样的问题。通常的解决方案是设置服务器的 `storage_engine` 选项：

```
storage_engine = blackhole
```

这只会影响没有显式定义存储引擎的 `CREATE TABLE` 命令。对于无法控制的现有应用程序，这种拓扑结构是脆弱的。可以禁用 InnoDB 并使用 `skip_innodb` 选项使表回归到 MyISAM，但是不能禁用 MyISAM 或 Memory 引擎。

另外一个主要的缺点就是使用某个从服务器代替主服务器。很难把从服务器提升为主服务器，因为分发主服务器使日志内容和坐标与原始的主服务器不同。

8.4.7 树或金字塔

如果正在把主服务器复制到大量的从服务器上，不管是把数据分发到不同的地方还是提供更多的读取性能，使用金字塔结构会更好管理，如图 8-12 所示。

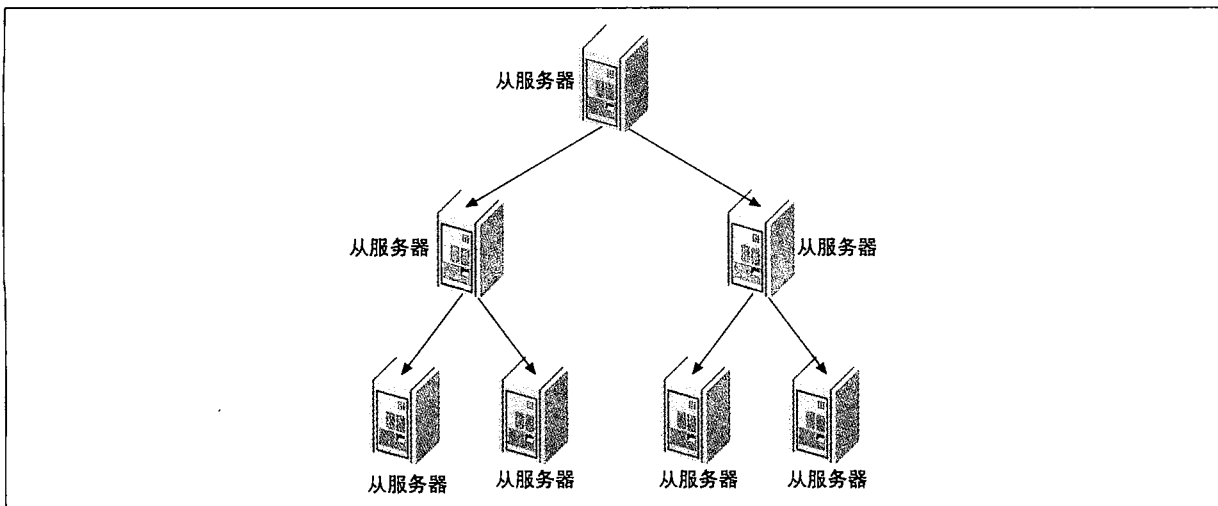


图 8-12: 金字塔形复制拓扑

371

这种设计的好处就是它减轻了主服务器的负担，就像上节的分布主服务器一样。它的缺点就是中间层出现的任何错误都会影响多个服务器，如果每个从服务器都直接和主服务器相连就没这个问题。同样，中间层次越多，处理错误就更困难，也更复杂。

8.4.8 定制复制解决方案

Custom Replication Solutions

MySQL 复制足够灵活，可以按照应用程序的需要定制复制的解决方案。典型的定制化方案是组合过滤、分发和向不同的存储引擎进行复制。还可以使用“黑客手段”，例如利用黑洞存储引擎（第 369 页“主服务器、分发服务器和从服务器”已经讨论过这种方式）。你的设计通常可以达到自己的要求，最大的局限是你自己的监控、管理及资源（网络带宽、CPU 功率等）。

选择性复制

为了利用访问局部性（Locality of Reference），并且使工作集停留在内存中，可以复制少量数据到从服务器。如果每个从服务器都有一部分主服务器的数据，并且读取被转向到从服务器，那么就可以更好地使用从服务器的内存。每个从服务器都只有一部分主服务器的写入负载，因此主服务器就会更加强大并不会使从服务器落在后面。

在某些方面，这种场景和我们下一章将会讨论的水平数据分区类似，但是它的优势在于主服务器包含了所有的数据。这意味着只需要为写入查询查找一台服务器，并且如果读取查询需要的数据不在从服务器上时，还可以通过主服务器进行查找。即使不能在从服务器上读取所有数据，也可以把很多读取从主服务器移走。

最简单的方式就是把数据分到不同的数据库中，然后把数据库移到不同的从服务器上。例如，如果你想把每一个部门的数据复制到不同的从服务器上，你可以创建名为 sales、marketing、procurement 等的数据库。每个从服务器都有一个叫 `replicate_wild_do_table` 的选项，它把数据限定在给定的数据库上。比如对于 sales 数据库，该选项为：

```
replicate_wild_do_table = sales.%
```

使用分发数据库进行过滤也是有用的。例如，如果想通过很慢或很昂贵的网络复制一台负载很重的服务器上的部分数据，就可以使用包含了黑洞表和过滤规则的本地分发服务器。

分发服务器可以使用过滤器从日志中移除不想要的内容。这可以在主服务器上避免危险的日志设置，并且不会把所有的日志通过网络传递到远程从服务器上。

分离功能

许多应用程序都是在线事务处理（OLTP）和在线分析处理（OLAP）的混合体。OLTP 查询较短并且是事务性的。OLAP 查询通常较大，也较慢，并且不会要求绝对的最新数据。这两种类型的查询都会给服务器增加不同的压力，因此，它们需要不同的配置，也许还需要不同的存储引擎和硬件才能得到最佳性能。

常见的解决方案是把 OLTP 服务器的数据复制到专门处理 OLAP 负载的从服务器上。这些从服务器有不同的硬件、配置、索引或不同的存储引擎。如果把从服务器专用于 OLAP 查询，那么就有可能允许更多的复制延迟或降低从服务器的服务质量。这也许意味着你可以用它运行非专属服务器不可接受的查询，例如执行时间很长的查询。

尽管你也许会选择忽略一些主服务器上的数据，但是也不需要进行特殊的复制设置。记住，仅仅需要用复制过滤器过滤掉一小部分数据也会减少 I/O 和缓存活动。

数据归档

可以在从服务器上归档数据，那就是说，通过在主服务器上执行删除查询并保证它不会在从服务器上执行，以把数据保存在从服务器上，并从主服务器上删除。有两种常见的办法可以做到：一是选择性地在主服务器上禁用二进制日志；二是在从服务器上使用 `replicate_ignore_db` 规则。

第 1 个方法要求执行 `SET SQL_LOG_BIN=0`，并且清理到主服务器上的数据。这种方式的好处在于不会要求从服务器进行特殊的配置，并且因为该命令不会被记录到主服务器的二进制日志中，它的效率也会稍微高一点。主要的缺点就是再也不能把主服务器上的二进制日志用于数据审核或按时间点恢复，因为日志没有包含对主服务器的数据做的任何修改。它还需要 `SUPER` 权限。

第 2 个办法是在执行清理数据的命令之前对特定的数据库使用 `USE`。例如，可以创建一个名为 `purge` 的数据库，然后再从服务器的 `my.cnf` 文件中定义 `replicate_ignore_db=purge`，并且重新启动从服务器。从服务器将会忽略对该数据库使用了 `USE` 的命令。这种方式没有第 1 个办法的缺点，但是它在从服务器提取不需要的二进制日志事件方面有一些小小的缺点。也可能有人在 `purge` 数据库上执行了非清理查询，从而导致从服务器无法重放该事件。

Makkit 的 `mk-archive` 工具支持这两种方式。



提示：第三种选择是使用 `binlog_ignore_db` 过滤复制事件，但是如同前文所述，它对于大多数情况都是危险的。

将从服务器用于全文搜索

许多应用程序要求合并事务和全文搜索。然而，只有 `MyISAM` 表提供了内建的全文搜索，并且 `MyISAM` 不支持事务。一种常见的规避方式就是配置一台从服务器，通过把某些表的存储引擎变为 `MyISAM` 实现全文搜索。

可以添加全文索引并在从服务器上执行全文搜索查询，这避免了在主服务器上使用了事务性和非事务性存储引擎的查询中可能出现的复制问题，并且它减轻了主服务器维护全文索引的工作。

只读从服务器

许多组织都喜欢只读从服务器，这样，意外的改变不会中断复制。可以使用 `read_only` 配置变量实现这个目标。它禁用了大部分写入，但也有一些例外，比如从服务器进程、有 `SUPER` 权限的用户和临时表。只要不给普通用户 `SUPER` 权限，就不会有问题。

模拟多主服务器复制

MySQL 现在并不支持多主服务器复制（一台从服务器有多台主服务器）。但是，可以通过把一台从服务器变成主服务器来模拟这种结构。例如，可以把从服务器指向主服务器 A 运行一段时间，然后把它指向主服务器 B 运行一段时间，最后再把它指向 A。这种办法的效果好坏取决于数据和两台主服务器在从服务器上引起的具体工作。如果主服务器负载较轻并且更新不会产生冲突，它就会工作得很好。

你需要做些工作来追踪每台主服务器的二进制日志坐标，还需要保证从服务器的 I/O 线程不会在每次循环中提取超过需要的数据，否则，网络流量就会因为反复的提取和抛弃大量数据显著上升。

374 一个可用的脚本可以从这儿下载：<http://code.google.com/p/mysql-mmre>。

也可以使用主—主（或者环形）结构，以及有从服务器的黑洞存储引擎来模拟多主服务器复制，如图 8-13 所示。

在这个配置中，每个主服务器都有自己的数据。它们也含有其他主服务器的表，但是利用黑洞引擎可避免在这些表中实际存储数据。从服务器和一个主服务器相连，不管是哪一个。这个从服务器根本不使用黑洞引擎，因此它对于两个主服务器都是高效的。

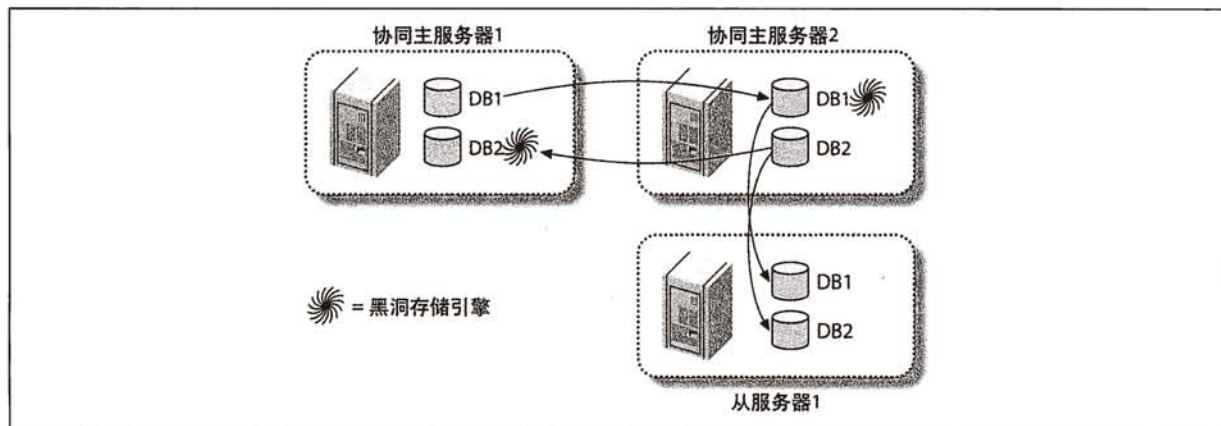


图 8-13：用双主服务器和黑洞存储引擎模拟多主服务器复制

实际上，并不真正需要使用主—主拓扑结构实现这个目标。可以简单地把 `server1` 和 `server2` 复制到从服务器。如果 `server2` 为从 `server1` 复制的表使用黑洞存储引擎，它就不会包含任何 `server1` 的数据，如图 8-14 所示。

这些复制都会遇到常见的问题，比如更新冲突和显式定义了存储引擎的 `CREATE TABLE` 命令。

创建日志服务器

使用 MySQL 的一种用途是创建没有数据的“日志服务器”，它的唯一目的是使其容易重放并且/或者过滤二进制日志事件。就像本章稍后所述，它对于崩溃后重启复制是很有用的。它也对按时间点恢复有用，具体内容参见第 11 章。

假设有一系列二进制日志和中继日志，也许来自于备份，也有可能来自于崩溃的服务器，并且你想在它们中间重放事件。可以使用 mysqlbinlog 提取事件，但是更方便和有效的方法是只创建一个没有任何数据的 MySQL 实例，并使它认为二进制日志是自己的。如果只是暂时需要日志服务器，可以使用 MySQL 沙盒脚本 (<http://sourceforge.net/projects/mysql-sandbox/>) 进行创建。日志服务器不需要任何数据，因为它不会执行日志，它只会提供日志给另外的服务器。(但是它不需要复制用户。)

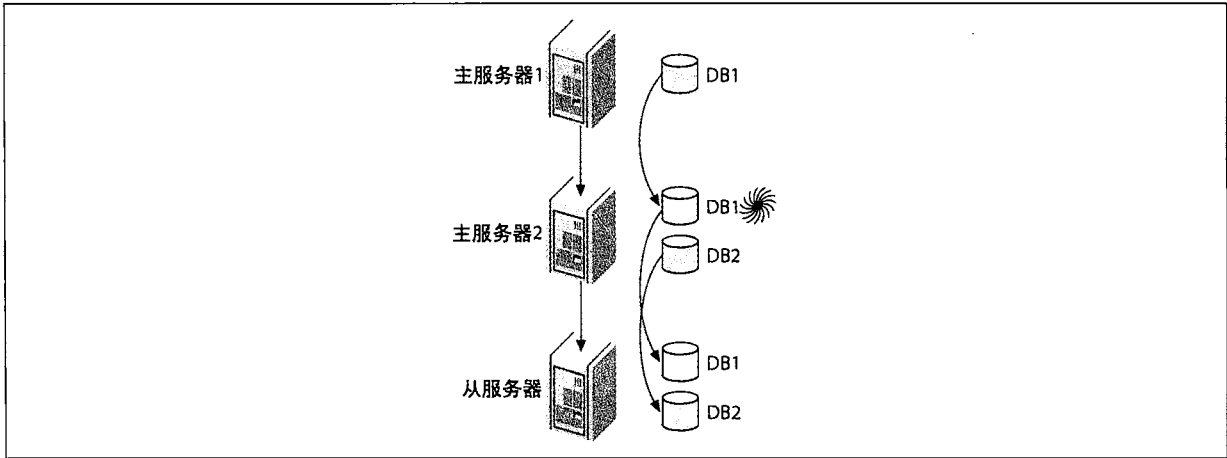


图 8-14：另外一种模拟多主服务器复制的例子

让我们看看这种策略是如何工作的（稍后会展示一些关于它的应用程序）。假设日志名为 somelog-bin.000001、somelog-bin.000002 等，把这些文件放到日志服务器的二进制日志目录。假设它为 /var/log/mysql，在启动日志服务器之前，按照下面的方式编辑 my.cnf。

```
log_bin          = /var/log/mysql/somelog-bin
log_bin_index    = /var/log/mysql/somelog-bin.index
```

服务器不会自动地发现日志文件，因此还需要更新服务器的日志索引文件。下面的命令可以在 Unix 系统上完成这个任务（注 9）：

```
# /bin/ls -l /var/log/mysql/somelog-bin.[0-9]* > /var/log/mysql/somelog-bin.index
```

确保运行在 MySQL 下的用户账户可以读写日志索引文件。现在可以启动日志服务器，并且使用 SHOW MASTER LOGS 验证它是否看见日志文件。

为什么从恢复这个角度来说，日志服务器要好于 mysqlbinlog？下面是一些原因：

- 它的速度更快，因为它不必从日志中提取命令并传递到 mysql。
- 可以容易地看到过程。

注 9：我们现实地使用了 /bin/ls 以避免启动通用别名，它们会为着色添加终端转义码。

- 可以容易地处理错误。例如，可以跳过不能复制的命令。
- 可以容易地过滤复制事件。
- 有时 mysqlbinlog 也许会因为日志格式更改而不能读取二进制日志。

8.5 复制和容量规划

Replication and Capacity Planning

写入通常是复制的瓶颈，并且很难使用复制扩展写入。在计划为系统添加多少从服务器的时候，要确保进行仔细的计算。这里很容易犯导致复制出问题的错误。

例如，假设工作负载中有 20% 写入和 80% 读取。为了计算简单，假设下面的情况都为真：

- 读和写都包含了同样的工作。
- 所有的服务器都完全一样，并且每秒进行 1 000 次查询。
- 从服务器和主服务器有同样的性能特征。
- 可以把所有的读取移到从服务器上。

如果当前有一个服务器每秒处理 1 000 个查询，那么应该添加多少从服务器，以使处理的负载加倍并把读取查询移到从服务器上？

看上去应该添加两台从服务器，并且在它们之间分摊 1 600 个读取查询。然而，不要忘记写入负载每秒已经增加到了 400，并且这不可能在主从服务器之间进行分摊。每个从服务器每秒都必须处理 400 个写入。这意味着写入查询占了 40%，并且每秒只能处理 600 个读取。因此，为了处理双倍负载，需要的不是 2 台而是 3 台从服务器。

如果负载再增加一倍呢？这时候每秒就会有 800 个写入，此时主服务器还可以处理，但是从服务器有 80% 的容量用于处理写入，因此需要 16 台从服务器来处理每秒 3 200 个读取查询。并且如果负载再增加一点，主服务器也变得不可承受负载了。

这远不是线性扩展，查询数量增加 4 倍，服务器的数量就会增加 17 倍。这意味着很快就会到达增加从服务器数量无法得到回报的地步。这仅仅是基于上面的假设，它忽略了一些事情，比如，单线程的基于命令的复制通常导致从服务器容量低于主服务器。真实的复制配置比这儿的理论计算还要差。

377 8.5.1 为什么复制无助于扩展写入

Why Replication Doesn't Help Scale Writes

很差的服务器容量比的根本原因就是不能把写入分摊到从服务器上。也就是说，服务器只能扩展读取，不能扩展写入。

你也许会疑惑是否有办法使用复制增加写入容量。答案是否定的，根本不行。对数据进行分区是唯一可以扩展写入的办法，下一章将会进行解释。

一些读者也许想到利用主—主拓扑（具体内容请参阅第 363 页“主动—主动模式下的主—主复制”）并写入两个

服务器。这种配置比主—从拓扑可以稍微增加一点写入，因为可以相同地在两个服务器之间共享串行化带来的损耗。如果在每台机器上执行 50% 写入，只有通过复制从另外的服务器上取得的 50% 查询需要被串行化。在理论上，这比在一台机器上对 100% 的写入进行并行处理，并且在从服务器上 100% 地进行串行处理划算。

这听上去较有吸引力。但是，这种配置处理的写入还及不上单台服务器。一台有 50% 的写入被串行化的服务器性能比一台全部查询都并行化的服务器低。

这是这种策略不能扩展写入的原因。它只是在两台服务器上共享串行写入的方法，所以“链中最弱的一环”不是那么弱了。它只提供了比主动—被动配置稍好一点的性能，但是增加了很多风险——并且它通常不能带来任何好处，具体原因见下一节。

8.5.2 计划成未充分使用

Plan to Underutilize

有意地让服务器不被充分使用是一个聪明并且划算的方式，尤其是在使用复制的时候。有多余容量的服务器可以更好地处理负载尖峰，有更多能力处理慢速查询和维护工作（比如 OPTIMIZE TABLE 操作），并且能更好地跟上复制。

试着利用主—主拓扑减少复制的问题通常不划算。对于这种拓扑，通常只能在每台服务器上加载小于 50% 的读取，因为如果增加了更多的负载，某台服务器又失效了，就没有足够的容量了。如果两台服务器都能自己处理负载，那么就不用担心复制问题了。

构建多余容量也是实现高可靠性的最佳方式之一，尽管有其他的方式，例如在有错误的时候在降级模式下运行应用程序。更多内容参阅下一章。

378

8.6 复制管理和维护

Replication Administration and Maintenance

创建复制也许不是经常会做的事情，除非有大量的服务器。但是它一旦就位，不管有多少服务器，对复制进行监控和管理就成了日常工作。

应该尽可能地把这个工作自动化。但是，你也不需要自己为这个工作写工具，在第 14 章，我们将讨论一些产品性的工具，需要有内建的复制监控功能。一些更有用的发布包括 Nagios、MySQL Enterprise Monitor 和 MonYOG。

8.6.1 监控复制

Monitoring Replication

复制增加了 MySQL 监控的复杂性。尽管复制实际发生在主从服务器上，大部分工作都在从服务器上完成，这是最常出问题的地方。所有的从服务器都正在复制？某个从服务器出错了？最慢的从服务器落后了多少？MySQL 提供了回答这些问题的大部分信息，但是自动化监控过程并且使复制更加健壮还是取决于你。

在主服务器上，可以使用 SHOW MASTER STATUS 命令查看主服务器当前二进制日志的位置和配置（参阅第 348 页“配置主服务器和从服务器”）。还可以询问主服务器哪个二进制日志在磁盘上：


```
mysql> SHOW MASTER LOGS;
+-----+-----+
| Log_name          | File_size |
+-----+-----+
| mysql-bin.000220  | 425605    |
| mysql-bin.000221  | 1134128   |
| mysql-bin.000222  | 13653     |
| mysql-bin.000223  | 13634     |
+-----+-----+
```

该信息在决定给予 PURGE MASTER LOGS 命令何种参数的时候有用。还可以使用 SHOW BINLOG EVENTS 命令在二进制日志中查看复制事件。例如，在运行前一个命令之后，我们在另一个不曾使用的服务器上创建一个表，我们知道命令在二进制文件中的偏移量是 13634，因此可以用下面的方式查看它：

379

```
mysql> SHOW BINLOG EVENTS IN 'mysql-bin.000223' FROM 13634\G
***** 1. row *****
  Log_name: mysql-bin.000223
    Pos: 13634
Event_type: Query
Server_id: 1
End_log_pos: 13723
  Info: use `test`; CREATE TABLE test.t(a int)
```

8.6.2 测量从服务器延迟

Measuring Slave Lag

最常见的事情之一就是需要测量从服务器到底落后于主服务器多少。尽管 SHOW SLAVE STATUS 输出的 Seconds_behind_master 栏从理论上显示了从服务器的延迟，事实上它并不总是准确的，原因如下：

- 从服务器计算 Seconds_behind_master 的方式是比较服务器当前的时间戳和二进制日志记录的时间戳，因此只有在处理查询的时候才能报告延迟。
- 如果从服务器进程没有运行，从服务器通常报告 NULL。
- 一些错误（比如，主、从服务器之间的 max_allowed_packet 设置不匹配，或者不稳定的网络）能中断复制，并且/或者终止从服务器线程，但是 Seconds_behind_master 只会报告 0，而不是指出错误。
- 即使从服务器进程正在运行，有时它也不能计算延迟。如果发生了这种情况，从服务器会报告 0 或 NULL。
- 非常长的事务能导致报告的延迟发生波动。例如，如果有事务更新了数据，在一小时内一直保持开启，然后进行提交，更新就会在实际发生之后的一小时才进入日志，它会短暂地报告比主服务器落后了一小时，然后延迟就很快变成 0。
- 如果分布主服务器落后了，并且有自己的从服务器，从服务器和分布主服务器保持了一致，即使相对于最终的服务器仍然有延迟，报告的值还是 0。

解决这些问题的办法就是忽略 Seconds_behind_master，并且使用可以直接观测和测量的工具来测量延迟。一个好的解决办法就是 heartbeat record，它是主服务器上每秒更新一次的时间戳。为了计算延迟，可以简单地用从服务器上的当前时间戳减去动率（Heartbeat）。这种方法没有前面谈到的那些问题，并且它还有一个好处就是可展示从服务器当前数据的状况。Maatkit 里面的 mk-heartbeat 脚本是复制动率的一个实现。

380

我们刚描述的延迟指标都不能表示从服务器到底需要多长时间才能赶上主服务器。这依赖于很多因素，例如从服务器的强大程度，以及主服务器要继续处理多少写入查询。

8.6.3 确定主、从服务器是否一致

Determining Whether Slaves Are Consistent with the Master



在完美的情况下，从服务器始终和主服务器的数据完全一样。但是在现实世界中，复制中的错误会导致从服务器的数据和主服务器的不一致。即使没有明显的错误，从服务器还是会因为 MySQL 不能正确复制的特性、MySQL 的缺陷、网络中断、崩溃、不妥当的关闭或者其他错误造成和主服务器不一致（注 10）。

我们的经验就是这其实是规则，而不是例外，它意味着检查主、从服务器之间的一致性应该是日常工作。如果把复制用于备份，这就是特别重要的事情，因为你不想从被破坏的从服务器上恢复数据。

本书的第一版中有一个用于比较主、从服务器表的行数的样例脚本。它肯定能揭示出一些区别，但是行的数量并不是相同数据的强有力的保证，真正需要的是比较表实际数据的有效方法。

MySQL 没有内建的方法来决定一台服务器的数据是否和另外的服务器完全一样。它提供了一些组件为表和数据生成校验和（Checksum），比如 CHECKSUM TABLE。但是，当复制正在进行的时候比较数据不是容易的事情。

Maatkit 有一个名为 mk-table-checksum 的工具解决了这个问题和其他的几个问题。这个工具有一些功能，包括同时快速地并行比较许多服务器，但是它的主要特性是可以校验从服务器的数据是否和主服务器一致，使用的方式是在主服务器上运行 INSERT ... SELECT 查询。

这些查询生成了数据的校验和，并把结果插入到了表中。该命令通过复制传递到了从服务器上并再次被执行。然后就可以把主服务器的数据和从服务器进行比较，并且查看数据是否不同。因为这个过程通过复制产生作用，它不用同时在两边都把表锁住。

使用该工具的典型方式就是在主服务器上运行它，使用和下面类似的参数：

```
$ mk-table-checksum --replicate=test.checksum --chunksize 100000 --sleep-coef=2  
<master_host>
```

该命令为所有表生成了校验和，每次大约处理 10 万行数据，并把结果插入 test.checksum 表。它每处理完一批数据就停顿一下，在处理最后一批数据的时候休眠两次，这保证查询不会阻塞正常的数据库操作。

一旦查询已经复制到从服务器，一个简单的查询就能检查主、从服务器的区别。Mk-table-checksum 能发现主服务器的从服务器，在所有的从服务器上运行查询并自动地输出结果。下面的命令限定了 10 层从服务器，从主服务器开始，并且打印出了和主服务器不同的表：

```
$ mk-table-checksum --replicate=test.checksum --replcheck 10 <master_host>
```

MySQL AB 计划在服务器自身实现类似的特性。这可能会好于外部脚本，但是在写作本书的时候，mk-table-checksum 是唯一的简单而可靠的比较工具。

8.6.4 从主服务器重新同步从服务器

Resyncing a Slave from its Master

你也许会不止一次地处理未被同步好的从服务器，也许是使用校验和发现了不同；也许是知道从服务器遗漏了查询，或者有人改动了从服务器上的数据。

注 10：如果正在使用非事务性的存储引擎，不首先使用 STOP SLAVE 就关闭服务器是不妥的。

传统的建议是停止从服务器并重新同步。如果不一致的从服务器是关键问题，那么也许就要在发现它的时候尽可能快地停止并移除它，然后从备份中重新克隆或恢复从服务器。

这种方法的缺点就是有些不便之处，特别是有大量数据的时候。如果可以发现不同的数据是哪些，那么就可以更高效地处理这个问题，而不是全部重新克隆一遍。如果发现的不一致并不重要，那么就可以把它保留在线上，然后重新同步受影响的数据。

最简单的修复是使用 `mysqldump` 转储并重新加载受影响的数据。在进行工作的时候，如果数据没有改变，那么这种办法会非常好。只需要简单地锁住主服务器上的表、转储表，等待从服务器赶上主服务器，然后在从服务器上导入表。（你需要等待从服务器跟上，从而就不会再引入更多的不一致。）

尽管这对许多场景都是可接受的解决方案，它通常不能工作在繁忙的服务器上。它也有在从服务器之外改动数据的缺点。通过复制改变从服务器的数据（通过在主服务器上做改动）通常是最安全的方法，因为它避免了竞争条件和其他出乎意料的事情。如果表很大或网络带宽有限，转储和重新加载的代价会很高。如果在一个有一百万行数据的表中只有序号为 1 000 的整数倍的行数据不同又会如何？转储和重新加载整个表在这种情况下是浪费的。

Mk-table-sync 是 Maatkit 的另外一个工具，它能解决一些问题。它可以高效地查找并解决表之间的不同。它也能通过复制进行操作，通过在主服务器上执行查询重新同步从服务器，因此不会有竞争条件出现。但是它不是在所有情况下都有效：为了正确地同步主、从服务器，它要求复制正在运行，因此在复制出错的时候它就不能工作。Mk-table-sync 的效率很高，但是对于及其大的数据不实际。在主、从服务器上比较 1TB 数据不可避免地会导致额外的工作。在那些它能工作的场景中，它能节省大量的时间和工作。

8.6.5 改变主服务器

Changing Masters

你早晚都会把从服务器指向新的主服务器，其原因也许是升级服务器，也许是主服务器出错且需要把从服务器变成主服务器，或者只是在重新分配容量。不管什么原因，你不得不改变主服务器。

如果这是计划好的工程，它就是容易的（或者至少比出紧急状况的时候更容易）。只需要简单地在从服务器上使用合适的值启动 `CHANGE MASTER TO` 命令。大部分值都是可选的，可以只定义要改变的值。从服务器会丢掉当前配置和中继日志，并且从新的主服务器进行复制。它也会利用新的参数更新 `master.info` 文件，因此改变将会不受从服务器重启的影响。

这个过程最难的一部分是弄明白新服务器上的位置，因此从服务器可以从上次停止的地方继续复制。

把从服务器变成主服务器更难一些。有两种情况需要用从服务器替换主服务器。第一个就是计划好的提升，另外一种则是非计划的提升。

计划的提升

把从服务器提升为主服务器在概念上是简单的。简单说来，有下面一些步骤：

1. 停止向老的主服务器写入。
2. 可选性地让从服务器赶上主服务器（这使下面的步骤更简单）。

3. 把一台从服务器配置为新主服务器。
4. 把从服务器和写入指向新主服务器，然后上面启动写入。

但是问题就藏在这些细节中。依赖于复制拓扑结构，一些场景是可能的。例如，主—主拓扑和主—从配置有一些不同。

更深入一点，下面是一些大多数需要配置的步骤：

1. 在当前主服务器上停止写入。如果可能，应该让所有客户端程序（不是复制连接）退出。它有助于用“没有运行”的标签构建客户端程序。如果使用虚拟 IP 地址，可以简单地关闭虚拟 IP，然后断开所有的客户端连接以关闭开放的事务。
2. 可选地利用 `FLUSH TABLES WITH READ LOCK` 停止所有的写入活动，还可以使用 `read_only` 选项把主服务器设置为只读。从这个观点来说，应该在即将被代替的主服务器上禁止所有的写入，因为一旦它不再是主服务器，写入就意味着丢失数据。
3. 选择某台从服务器成为主服务器，并且保证它完全能赶上复制（例如，让它把从主服务器提取的中继日志执行完）。
4. 可选地验证新主服务器和旧主服务器数据是否相同。
5. 在新主服务器上执行 `STOP SLAVE`。
6. 在新主服务器上执行 `CHANGE MASTER TO MASTER_HOST='',` 然后执行 `RESET SLAVE`，使其断开和老服务器的连接并且丢掉 `master.info` 文件里面的连接信息。（如果连接信息在 `my.cnf` 中定义了，那么这种方式就无法工作，这也是我们推荐不要把连接信息放在这个文件中的一个原因。）
7. 使用 `SHOW MASTER STATUS` 了解新主服务器二进制日志的坐标。
8. 保证所有其他的从服务器能赶上。
9. 关闭旧的主服务器。
10. 在 MySQL 5.1 和更高版本中，如果需要，则在新主服务器中激活事件。
11. 将客户端连接到新服务器。
12. 在每个从服务器上启动 `CHANGE MASTER TO`，指向新主服务器，使用从 `SHOW MASTER STATUS` 中得到的坐标。



提示：在把从服务器提升为主服务器的时候，要保证移除任何从服务器特有的数据库、表和权限。还需要更改任何从服务器特定的参数，例如 `innodb_flush_log_at_trx_commit` 选项。同时，如果把主服务器降级，要保证可进行需要的配置。

如果主、从服务器的配置完全一样，那就不用更改任何东西。

未计划的提升

如果主服务器崩溃了，不得不把从服务器提升起来，这个过程可能就不那么容易了。如果只有一个从服务器，只需要使用从服务器；但是如果有更多的从服务器，就需要一些额外的步骤完成提升。

这种情况还加剧了丢失复制事件的问题。有可能一些已经发生在主服务器上的更新还没有被复制到从服务器上，甚至还可能一个命令回滚了主服务器，但是还没有回滚从服务器，因此从服务器的实际位置会超过主服务器的逻辑位置。

辑复制位置（注 11）。如果能在某个时候恢复主服务器的数据，也许就能取得丢失的命令并且手动地应用它们。

在下面的所有步骤中，确保在计算中使用 `Master_log_File` 和 `Read_Master_Log_Pos` 的值。下面是把从服务器从主—从拓扑中提升为主服务器的过程：

1. 决定哪个从服务器有最新数据。在每个从服务器上检查 `SHOW SLAVE STATUS` 的输出，并且选择 `Master_Log_File/Read_Master_Log_Pos` 的坐标是最新的那个服务器。
2. 让所有的从服务器执行完在旧主服务器崩溃之前提取的所有中继日志。如果在从服务器执行完中继日志之前就改变主服务器，它就会抛弃剩下的日志事件，我们就不知道它在什么地方停止了。
3. 执行前一节中的 5~7 步。
4. 比较每个从服务器的 `Master_Log_File/Read_Master_Log_Pos` 坐标。
5. 执行前一节中的 10~12 步。

我们假设在所有的从服务器上都开启了 `log_bin` 和 `log_slave_updates`。开启这些选项使你可以把所有从服务器恢复到一致的时间点，如果没有则不能可靠地做到这一点。

确定想要的日志位置

如果任何从服务器都和新服务器的位置不同，那么就要按照从服务器复制的最后一个事件找出新主服务器的二进制日志的位置，并且把它用于 `CHANGE MASTER TO`。可以使用 `mysqlbinlog` 工具来检验从服务器执行的最后一个查询，并且在新主服务器的二进制日志中找到同样的查询。通常还需要做一点小小的数学运算。

385 为了显示这个过程，假设日志事件有递增的 ID 号，并且最新的从服务器——新主服务器——在旧主服务器崩溃的时候只接收到了 ID 号为 100 的事件。现在假设有两台从服务器，`slave2` 和 `slave3`，`slave2` 收到了第 99 个事件，`slave3` 收到了第 98 个事件。如果两个从服务器都指向新主服务器当前的二进制日志内容，它就会从第 101 个事件开始复制，因此它们的内容就不同步了。如果新服务器的二进制日志使用了 `log_slave_updates`，就可以在新主服务器的二进制日志中找到第 99 个和第 100 个事件，从服务器就可以保持一致。

由于服务器重启、不同的配置、日志轮转或 `FLUSH LOGS` 命令，同样的事件在不同的服务器上的位置会有差异。查找事件可能会既慢又让人生厌，但是通常不是很难。只需要使用 `mysqlbinlog` 从二进制日志或中继日志中检查每个从服务器上最后执行的事件，然后还是使用 `mysqlbinlog` 在新主服务器的二进制日志中找到同样的查询，它就会打印出查询的偏移量，最后就可以在 `CHANGE MASTER TO` 查询中使用这个偏移量。

可以通过把新主服务器和从服务器的字节偏移进行相减使该过程更快，它显示了字节位置的差异。如果把这个值和新主服务器的当前二进制日志位置相减，很可能会得到想要的查询位置。你只需要验证它，就会发现需要用来启动从服务器的位置。

让我们看一个例子。假设 `server1` 是 `server2` 和 `server3` 的主服务器，它崩溃了。按照 `SHOW SLAVE STATUS` 中的 `Master_Log_File/Read_Master_Log_Pos`，`server2` 已经成功地复制了 `server1` 的二进制日志中的事件，但是 `server3` 不是最新数据。图 8-15 显示了这种场景（日志事件和字节偏移仅仅出于展示目的）。

如图 8-15 所示，我们可以确定 `server2` 复制了所有的事件，因为它的 `Master_Log_File/Read_Master_Log_Pos`

注 11：这实际是可能的，尽管 MySQL 只有在事务提交的时候才记录日志。更多细节请参阅本章稍后第 391 页“混合事务性和非事务性表”。

和 server1 最后一个位置相匹配。因此，可以把 server2 提升为新的主服务器，并且使 server3 成为它的从服务器。

但是 server3 应该对 CHANGE MASTER TO 使用什么参数？这需要做一些数学运算和调查。Server3 停止在偏移 1 493 处，它比偏移 1582 落后了 89 个字节，偏移 1582 是 server2 执行的最后一个命令。Server2 当前把位置 8 167 写入二进制日志中。8167-89=8078，因此在理论上需要在 server2 的日志中指出 server3 的偏移。调查该位置附近的日志事件，并且验证 server2 的日志中该偏移处确实有该事件是个好主意。但是也有可能有的事情发生，因为数据更新仅仅发生在 server2 上。

假设事件都是一样的，下面的命令将会把 server3 切换为 server2 的从服务器：

```
server2> CHANGE MASTER TO MASTER_HOST="server2", MASTER_LOG_FILE="mysql-bin.000009",  
MASTER_LOG_POS=8078;
```

如果服务器实际已经完成了执行并且记录了更多的事件，偏移量超过了 1 582，但是它崩溃了将会如何？因为 server2 仅仅读取并执行到偏移 1 582，就可能会永远丢失事件。但是，如果旧主服务器的磁盘没有被破坏，你还能用 mysqlbinlog 或日志服务器从二进制日志中恢复丢失的事件。

如果需要从旧主服务器恢复丢失的事件，我们推荐在提升了新主服务器之后，在客户端连接之前进行恢复。使用这种方式，你就不用在每个从服务器上执行丢失的事件，复制将会帮你做到这一点。但是如果失效了的主服务器完全不可用，你就不得不等待，稍后再完成这项工作。

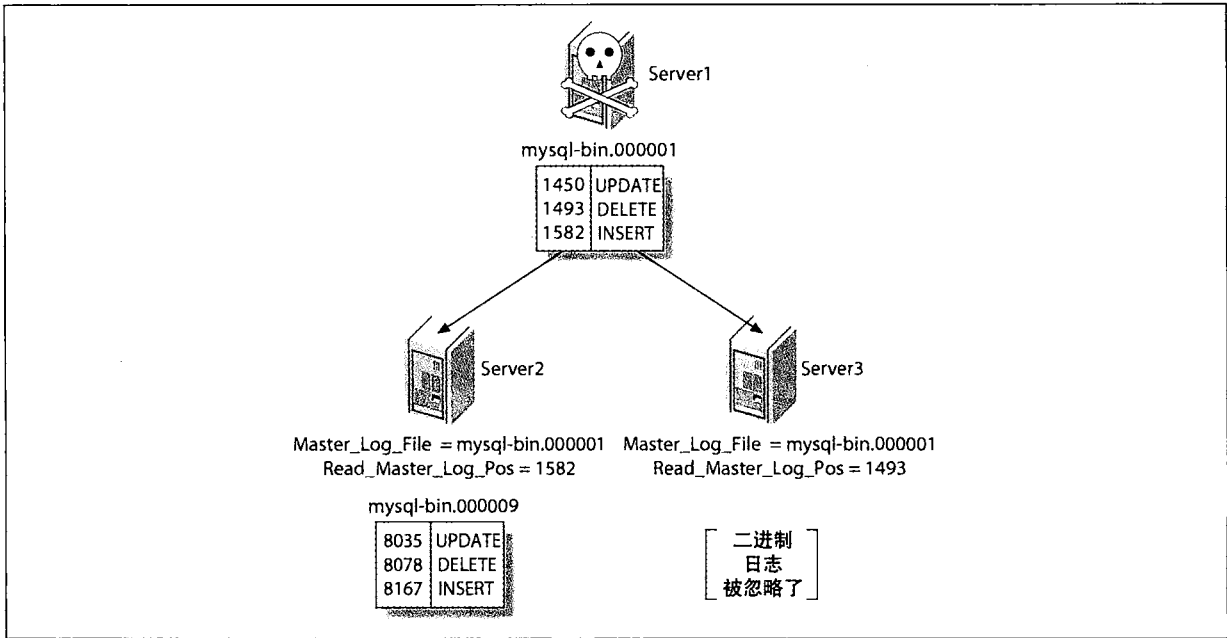


图 8-15：当 Server1 崩溃，Server2 跟上了 Server1，但是 Server3 在复制中落后了

这个过程的一种变化是使用可靠的方式存储主服务器的二进制日志文件，比如 SAN 或分布式复制数据块设备 (Distributed Replicated Block Device DRBD)。即使主服务器完全失效了，你还是会有二进制日志文件。可以创建一个日志服务器，把从服务器指向它，然后让它们跟上主服务器失效的点。这使其中一个从服务器提升为新主服务器变得不再重要——本质上，这与计划提升的过程相同。我们在下一章进一步讨论这些存储选项。



警告：当把从服务器提升为主服务器的时候，不要改变它的服务器 ID，以和旧主服务器相匹配。如果这么做了，就不能使用日志服务器在旧主服务器上重放事件。这是固定服务器 ID 是好主意的一个原因。

8.6.6 在主—主配置中交换角色

Switching Roles in a Master-Master Configuration

主—主复制的一个好处就是可以方便地交换主动和被动角色，因为它们的配置是对称的。在这节，我们展示如何完成交换的方法。

当在主—主配置中交换交换角色的时候，最重要的事情就是保证在任何时候只有一个主服务器被写入。如果向主服务器的写入发生了交叉，写入就会发生冲突。换句话说，被动服务器在交换角色之后就不能再从主动服务器接收任何二进制日志事件。可以通过确保被动服务器的从服务器线程在可写入之前跟上主动服务器来保证这样的事情不会发生。

下面的步骤交换了服务器角色，但是不会导致任何冲突：

1. 在主动服务器上停止所有的写入。
2. 在主动服务器上执行 `SET @@global.read_only := 1`，并且在它的配置文件中设置 `read_only` 选项。要知道，这将不会防止具有 `SUPER` 权限的用户更改数据。如果想防止所有用户更改数据，就使用 `FLUSH TABLES WITH READ LOCK`。如果没这么做，就必须断开所有客户端连接，以保证没有长时间运行的命令和未提交的事务。
3. 在主动服务器上执行 `SHOW MASTER STATUS`，并且注意二进制日志的坐标。
4. 在被动服务器上使用主动服务器的二进制日志坐标执行 `SELECT MASTER_POS_WAIT()`，这个命令将会在从服务器进程跟上主动服务器之前执行阻塞动作。
5. 在被动服务器上执行 `SET @@global.read_only := 0`，这使其成为主动服务器。
6. 重新配置应用程序，使它写入新主动服务器。

依赖于应用程序的配置，你也许还需要进行其他的操作，包括在两台服务器上改变 IP 地址。我们在下一章讨论该问题。

8.7 复制问题和解决方案

Replication Problems and Solutions

中断 MySQL 的复制并不是难事。它的实现方式很简单，这意味着配置很容易，但也意味着有很多方式可以停止它，把它弄混乱甚至中断。这里列出了常见的问题，探究如何揭示问题，如何解决问题，甚至防止问题。

8.7.1 导致数据损坏或丢失的错误

Errors Caused by Data Corruption or Loss

出于不同的原因，MySQL 复制并不能很容易地从崩溃、断电和由磁盘、内存或网络错误导致的破坏中恢复，基

本上可以肯定有时会因为这些问题而导致复制重启。

大部分在意料之外关闭服务器后出现的问题都来自一台服务器没有把数据刷写到磁盘上。下面是意外关闭服务器时会出现的一些问题。

主服务器意外关闭

如果主服务器没有使用 `sync_binlog` 配置主服务器，它可能就不会在崩溃之前把二进制日志事件写到磁盘上，从服务器的 I/O 线程因此也会处于读取永远不会到达磁盘的事件的过程中。当主服务器重新启动时，从服务器将重新连接并且尝试再次读取事件，但是主服务器的响应无法找到该事件。二进制日志转储过程通常是同时的，因此这种情况很常见。

解决这个问题的方法是指导从服务器从下一个二进制日志的开始部分读取。然而，一些日志事件将会永久丢失，可以通过配置主服务器的 `sync_binlog` 防止这种事情。

即使已经配置了 `sync_binlog`，MyISAM 在崩溃的时候还是会被损坏，如果 `innodb_flush_logs_at_trx_commit` 没有被设置为 1，InnoDB 也会出现这种问题。

从服务器意外关闭

当从服务器在意外关闭之后重新启动，它会读取 `master.info` 文件以决定它停止复制的地点。不幸的是，该文件没有被同步到磁盘，因此它包含的信息可能是错误的。从服务器将可能尝试重新执行一些二进制日志事件，这可能会导致某些独特的索引损害。除非能确定日志停止的确切地方，如果不可能，那么就只能跳过它引发的错误。Mk-slave-restart 工具有助于做到这一点。

如果使用 InnoDB 表，可以在重启从服务器之后查看 MySQL 错误日志。InnoDB 恢复过程中打印出了二进制日志的坐标，并且可以用它们来确定从服务器指向主服务器的位置。

除了意外关闭 MySQL 导致的数据丢失，磁盘上的二进制日志或中继日志损坏也是常见的。下面是一些更常见的情景。

主服务器上的二进制日志损坏

如果主服务器上的二进制日志损坏，那么除了跳过损坏的部分就别无选择。可以在主服务器上运行 `FLUSH LOGS`，这样它就会创建一个新的日志文件，并且让从服务器指向新的日志，也可以试着去发现损坏区域的结尾。有时可以使用 `SET GLOBAL SQL_SLAVE_SKIP_COUNTER = 1` 来跳过单个损坏的事件。如果损坏的事件多于一个，就不停地重复该过程，直到跳过所有的事件。

从服务器上中继日志损坏

如果主服务器的二进制日志未受损坏，可以使用 `CHANGE MASTER TO` 丢掉并重新提取中继日志。此时只需要把从服务器指向最近复制的位置即可 (`Relay_Master_Log_File/Exec_Master_Log_Pos`)，这会导致从服务器丢掉磁盘上的所有中继日志。

二进制日志和 InnoDB 事务日志不同步

如果主服务器崩溃，即使事务没有被写入磁盘上的二进制日志，InnoDB 也会把它记录为已提交。除非它在从服务器的中继日志中，否则就无法恢复它。在 MySQL 5.0 中，可以使用 `sync_binlog` 防止这一事件，在 MySQL 4.1 中，则可以使用 `sync_binlog` 和 `safe_binlog`。

当二进制日志损坏了，能恢复多少数据取决于崩溃的类型。下面是几种常见的类型。

数据改变，但是事件仍然有效

不幸的是，MySQL 甚至不能检测到该类型的损坏。这表明了经常检查从服务器数据是个好主意。

数据改变并且事件无效

也许可以使用 `mysqlbinlog` 提取事件并看到垃圾数据，例如：

```
UPDATE tbl SET col????????????????
```

试着查找下一个事件的开头并把它打印出来，可以通过添加偏移和长度做到这一点。这样也许就可以跳过该事件。

数据已跳过并且/或者事件的长度错误

在这种情况下，`mysqlbinlog` 有时会因为无法读取事件并且无法找到下个事件的开头而报错或崩溃。

一些事件已经损坏或被重写，或者偏移已经发生了改变并且下一个事件位于错误的偏移

和上面一样，`mysqlbinlog` 还是会出错。

390 当损坏足够严重，`mysqlbinlog` 不能读取事件，那么就不得不进行十六进制的编辑和一些繁琐的工作以找到事件的范围。这通常并不困难，因为事件之间的分隔符是可读的。

下面有一个例子。首先，使用 `mysqlbinlog` 检查样例日志中的事件偏移：

```
$ mysqlbinlog mysql-bin.000113 | egrep '^# at '
```

```
# at 4
# at 98
# at 185
# at 277
# at 369
# at 447
```

一种发现偏移的简单办法就是把偏移和下面的 `strings` 命令的输出做对比：

```
$ strings -n 2 -t d mysql-bin.000113
```

```
1 binpC'G
25 5.0.38-Ubuntu_0ubuntu1.1-log
99 C'G
146 std
156 test
161 create table test(a int)
186 C'G
233 std
243 test
248 insert into test(a) values(1)
278 C'G
325 std
335 test
340 insert into test(a) values(2)
370 C'G
417 std
427 test
432 drop table test
448 D'G
474 mysql-bin.000114
```

有很明显的模式可以定位事件的开头。注意到以 'G 结尾的字符串就位于事件开头的一字节之后。它们是固定长度的事件头的一部分。

具体的值在不同的服务器上会不一样。经过检查，就能发现二进制日志的模式，取得日志事件的偏移。接下来就可以给 `mysqlbinlog` 应用 `--start-position` 参数，或者为 `CHANGE MASTER TO` 命令使用 `MASTER_LOG_POS` 参数以跳过出错的事件。

8.7.2 使用非事务性表

如果一切正常，基于命令复制通常能很好地处理非事务性表。然而，如果在更新非事务性表时出了错误，例如命令在完成之前就被结束掉了，主、从服务器最终就会有不同的数据。

例如，假定更新一个有 100 行数据的 MyISAM 表。如果命令正好在更新了 50 行的时候被结束掉了，会发生什么情况？有一半行被更新了，但是另外一半没有。复制会导致不同的数据，因为命令会在从服务器上重放，它会改变从服务器的 100 行数据。（MySQL 随后会注意到命令在主服务器上引发了一个错误，但是从服务器没有，此时复制就会出错并停止。）

如果使用 MyISAM 表，要确保在停止 MySQL 服务器，或者会结束任何正在运行的查询（包括任何未完成的更新命令）的关闭服务器之前运行 `STOP SLAVE` 命令。事务性存储引擎没有这样的问题。如果正在使用事务性表，主服务器上失败的更新将被回滚，并且不会被记录到二进制日志中。

8.7.3 混合事务性和非事务性表

当你使用事务性存储引擎，MySQL 只有在事务提交的时候才会把命令记录到二进制日志中。因此，如果事务被回滚了，MySQL 就不会记录命令，因此在从服务器上就不会进行重放。

然而如果混合了事务性和非事务性表，并且发生了回滚，MySQL 能回滚事务性表，却无法回滚非事务性表。只要不发生类似于更新中途被结束掉这样的错误，就不会有问题：MySQL 此时不会只记录命令，还会记录一个 `ROLLBACK` 命令。这样的结果就是同样的命令可以在从服务器上执行，所有的情况都很正常。这缺乏一些效率，因为从服务器必须做一些工作然后再把它们丢掉，但从服务器将在理论上和主服务器保持同步。

到目前为止，一切都很正常。问题在于从服务器上发生了死锁，但是主服务器没有发生。使用事务性存储引擎的表将在从服务器上回滚，但是从服务器不能回滚非事务性表。这样，主、从服务器的数据就会不一样。

防止这个问题的唯一办法就是避免混合使用事务性和非事务性表。如果遇到了这种问题，唯一的办法就是跳过从服务器上的错误，并且重新同步相关的表。

从原则上说，基于行复制不会有这个问题。基于行复制记录了行的改变，而不是 SQL 命令。如果命令改变了 MyISAM 表中的某些行，同时主服务器的 InnoDB 表发生了死锁并且进行了回滚，MyISAM 表的更改将被记录到日志中并在从服务器上进行重放。我们测试了一些简单的例子，发现它能很正常地工作。然而，在写作本书的时候，我们还没有足够的经验证明基于行复制是否能完全避免混合事务性表和非事务性表的问题。

8.7.4 不确定命令

任何以非确定方式更改数据的命令都会导致主、从服务器数据不一致。例如，使用了 `LIMIT` 的 `UPDATE` 命令依

依赖于命令发现表中行的顺序。除非能确保顺序在主、从服务器是完全一致的，例如，行按照主键进行排序，否则就会导致主、从服务器上更改的数据不一致。这种问题很难被注意到，因此一些人制定规则，不允许在更改数据的命令中使用 `LIMIT`。

也要注意涉及 `INFORMATION_SCHEMA` 表的命令。它在主、从服务器上很容易就变得不一致，因此结果也会不同。最终，要注意许多服务器变量，比如 `@@server_id` 和 `@@hostname`，在 MySQL 5.1 之前的版本中不能被正确地复制。

基于行复制没有这些局限。

8.7.5 主、从服务器使用不同的存储引擎

Different Storage Engines on the Master and Slaves

从服务器上使用不同的存储引擎是很方便的事情。但是，在某些情况下，基于命令复制会因为不同的存储引擎产生不同的结果。例如，非确定性命令（例如上节中提到的命令）在存储引擎不同的情况下更容易导致问题。

如果发现主、从服务器上特定表的数据不一致，就应该检查服务器使用的存储引擎及更新数据的命令。

8.7.6 从服务器上数据改变

Data Change on the Slave

基于命令复制依赖于主、从服务器有一样的数据，因此不需要在从服务器上进行任何数据更改（较好的办法是使用 `read_only` 配置选项）。考虑下面的命令：

```
mysql> INSERT INTO table1 SELECT * FROM table2;
```

393 如果从服务器的 `table2` 有不同的数据，`table1` 也会有不同的数据。换句话说，数据更改会从一个表传递到另一个表。这对所有的查询类型都有效，不仅仅是 `INSERT ... SELECT` 查询。这会导致两种可能的结果：从服务器上发生索引冲突或根本没有任何错误。有错误会好一些，因为至少还能收到警告，表明主、从服务器数据不一致。不可见的不同会悄悄地造成灾难。

解决该问题的唯一办法是重新同步主服务器的数据。

8.7.7 服务器 ID 不唯一

Non Unique Server IDs

这个问题更加难以捉摸。如果偶然地给两个从服务器设置了同样的服务器 ID，那么它们看上去不会有什么问题。但是如果检查错误日志，或者使用 `innotop` 查看主服务器，那么就会看到一些很奇怪的事情。

在主服务器上，在任何时候都只会看到一台从服务器的连接。（通常，所有的从服务器都是连接的，并且随时都在进行复制。）在从服务器上的错误日志中，会经常看到断开连接和重新连接的错误信息，但是不会提及服务器 ID 没有被配置好。

随着 MySQL 版本不同，复制可能会缓慢地正确进行，也可能会不能正常工作：给定的从服务器可能会丢失二进制日志事件，或者重复执行事件，造成重复键错误（或者不可见的数据库损坏）；也可能会因为从服务器之间彼

此竞争造成主服务器负载增加，导致数据损坏或崩溃。如果从服务器竞争得非常厉害，错误日志就会在很短的时间内快速增加。

唯一解决的办法就是仔细地设置从服务器。一个较好的办法是创建一个主、从服务器 ID 的映射表，这样就可以跟踪到 ID 的信息（注 12）。如果从服务器都在同一个子网里面，那么就可以使用机器 IP 地址的最后八位作为唯一的 ID。

8.7.8 不确定的服务器 ID

Undefined Server IDs

如果没有在 my.cnf 文件中定义服务器 ID，那么可以使用 CHANGE MASTER TO 创建复制，但是无法启动从服务器：

```
mysql> START SLAVE;  
ERROR 1200 (HY000): The server is not configured as slave; fix in config file or with  
CHANGE MASTER TO
```

如果刚刚使用了 CHANGE MASTER TO，并且使用 SHOW SLAVE STATUS 验证了设置，这个错误尤其让人困惑。使用 SELECT @@server_id 取得的值只是默认值。你不得不显式地设置这个值。

8.7.9 对未复制数据的依赖性

Dependencies on Nonreplicated Data

如果主服务器上的数据库或表在从服务器上不存在，那么就很容易破坏复制。假设主服务器上有一个从服务器没有的数据库，如果主服务器上的任何数据更新引用了该数据库中的表，当从服务器重放查询的时候，复制就会被中断。

这个问题没办法解决，唯一的办法就是避免在主服务器上创建从服务器没有的表。

这种表是如何创建出来的？有很多种方式，其中的一些方式比其他方式更难防止这个问题。例如，假设你在从服务器上创建了一个数据库，但是主服务器上没有，随后交换了主、从服务器角色。当交换的时候，你也许会忘记删除这个数据库的相关权限，现在有人连接到新主服务器并在那个服务器中运行查询，或者周期性的工作会发现这些表并对每一个表运行 OPTIMIZE TABLE。

这是在把从服务器变为主服务器，或者决定如何配置从服务器时需要记住的一件事情。任何导致主、从服务器不一致的事情，都有可能将来引发问题。

基于行复制也许会解决一部分问题，但是现在下这样的结论还为时过早。

8.7.10 丢失的临时表

Missing Temporary Tables

临时表的某些用途是很方便的，但是不幸的是它们和基于命令复制不兼容。如果从服务器崩溃或关闭了，那么任何从服务器线程使用的临时表都消失了。在重启从服务器的时候，任何引用了丢失的临时表的命令都会失败。

注 12：也许你想把它保存在数据库中？这并不是完全在开玩笑，可以给 ID 列添加一个唯一索引。

使用基于命令复制，在主服务器上没有使用临时表的安全办法。许多人都很喜欢临时表，因此很难说服他们，但这是真的。不管临时表存在的时候有多短暂，它都有可能让从服务器无法恢复。即使在单个事务内部使用，也会有这样的问题。（在从服务器上使用临时表出问题的可能性会比较小，但是如果从服务器成了主服务器，该问题还是会存在。）

如果复制因为从服务器重启之后无法发现临时表而停止，那么就确实有一些工作需要做。可以跳过遇到的错误，或者手动创建一个和临时表有同样的名字和结构的表。不管用什么办法，如果写入查询引用了临时表，数据都有可能与从服务器不一致。

消灭临时表其实并不是那么困难。临时表最有用的两个特性是：

- 它们只对创建它们的连接可见，因此它们不会和其他连接的同名临时表冲突。
- 当连接关闭的时候，它们就消失了，因此不用显式地移除它们。

可以在数据库中创建持久表，把它们作为伪临时表，以模拟这些特性。你需要为它们选择唯一的名字。幸运的是，这是容易办到的事情：简单地把连接 ID 拼接到表后面就可以了。例如，在执行 `CREATE TEMPORARY TABLE top_users(...)` 的地方，就可以执行 `CREATE TABLE temp.top_users_1234(...)`，这儿的 1234 是 `CONNECTION_ID()` 返回的值。在应用程序使用完伪临时表之后，可以把它们删掉或利用清理进程移除它们。表名中含有连接 ID，这样就能很容易地决定哪个表不再有用——可以用 `SHOW PROCESSLIST` 得到所有的活动连接，然后和表名中的连接 ID 进行比较（注 13）。

使用真正的表代替临时表也有其他的一些好处。例如，它使调试程序变得容易，因为可以看到应用程序正在操纵来自于其他连接的数据。如果使用临时表，就不能轻易地做到这一点。

但是真实表有一些临时表没有的开销：创建真实表的速度较慢，因为需要把 .frm 文件同步到磁盘上。可以关闭 `sync_frm` 选项来加速这一动作，但是这比较危险。

如果使用临时表，就应该在关闭从服务器之前确保 `Slave_open_temp_tables` 状态变量是 0。如果它不是 0，那么就可能在重启从服务器的时候出问题。正确的方法就是运行 `STOP SLAVE`，检查变量值，然后再关闭从服务器。如果在停止从服务器进程之前检查该变量，那么就有发生竞争条件的风险。

8.7.11 不复制所有的更新

Not Replicating All Updates

如果错误使用了 `SET SQL_LOG_BIN=0`，或者没有理解复制过滤器的规则，从服务器也许就不会执行某些发生在主服务器上的更新。有时你想利用这个进行归档，但是它通常都会发生意外并有坏的结果。

例如，假设有一个 `replicate_do_db` 规则，把 `sakila` 数据库复制到某一台从服务器上。如果在主服务器上执行下面的命令，从服务器的数据就会和主服务器不一样：

```
mysql> USE test;
mysql> UPDATE sakila.actor ...
```

其他类型的命令甚至会因为未被复制的依赖性而引起复制失败。

注 13: `mk-find` 是 `Maatkit` 中的另外一个工具，它可以使用 `--pid` 和 `--sid` 选项轻易地移除伪临时表。

8.7.12 InnoDB 加锁读取导致的锁争用 (Lock Contention)

Lock Contention Caused by InnoDB Locking Reads

InnoDB 的 SELECT 命令通常是非锁定的，但是在某些情况下需要锁，尤其是 INSERT ... SELECT 默认会锁定它读取的所有行。MySQL 在从服务器上执行命令的时候需要锁，以保证得到完全一样的结果。实际上，锁定在主服务器上串行了命令，它匹配了从服务器执行命令的方式。

这种设计可能会遇到锁竞争、阻塞和锁定等待超时。一种缓解的办法就是不要让事务在不需要的时候也保持开启，这会导致较少的阻塞。可以在主服务器上尽快地提交事务释放掉锁。

把大命令拆成若干个小命令，使它尽可能地简短，也会对此有帮助。这是一种减少锁争用非常有效的方式，即使有时很难做到这一点，它也是值得的。

另外一种变通的办法是用 SELECT INTO OUTFILE，再加上 LOAD DATA INFILE 来代替 INSERT ... SELECT 命令。这样不仅快，而且不会要求锁定。它是一种特殊的手段，但是有时还是有用的。最大的问题是为输出文件选择一个唯一的名字，以及在完成的时候清理掉文件。可以使用前面第 394 页“丢失临时表”中提到过的 CONNECTION_ID() 办法来解决文件名唯一的问题，并且可以使用周期性工作（Unix 中的 crontab、Windows 中的计划任务）来清理掉已经使用过的输出文件。

你也许想关闭锁定，而不是利用这些变通的方法。你可以这么做，但是对于大多数场景，这都不是一种好办法，因为它会使从服务器不知不觉间就和主服务器不同步了。它也会导致恢复服务器的时候，二进制日志失去作用。但是，如果觉得这么做的好处大于坏处，可以用下面的方式来关闭锁定：

```
innodb_locks_unsafe_for_binlog = 1
```

这使命令的结果依赖于没有锁定的数据。如果第 2 个命令修改了数据，并且在第 1 个命令完成之前进行了提交，那么在重放二进制日志的时候两个命令可能就无法产生同样的结果。这对于复制和按时间恢复都是正确的。

为了了解锁定读取是如何防止混乱的，想象一下有两个表：一个没有数据，另外一个只有一行，数据为 99。两个事务更新了数据，事务 1 把第 2 个表中的数据插入第一个表里，并且事务 2 更新了第 2 个表，如图 8-16 所示。

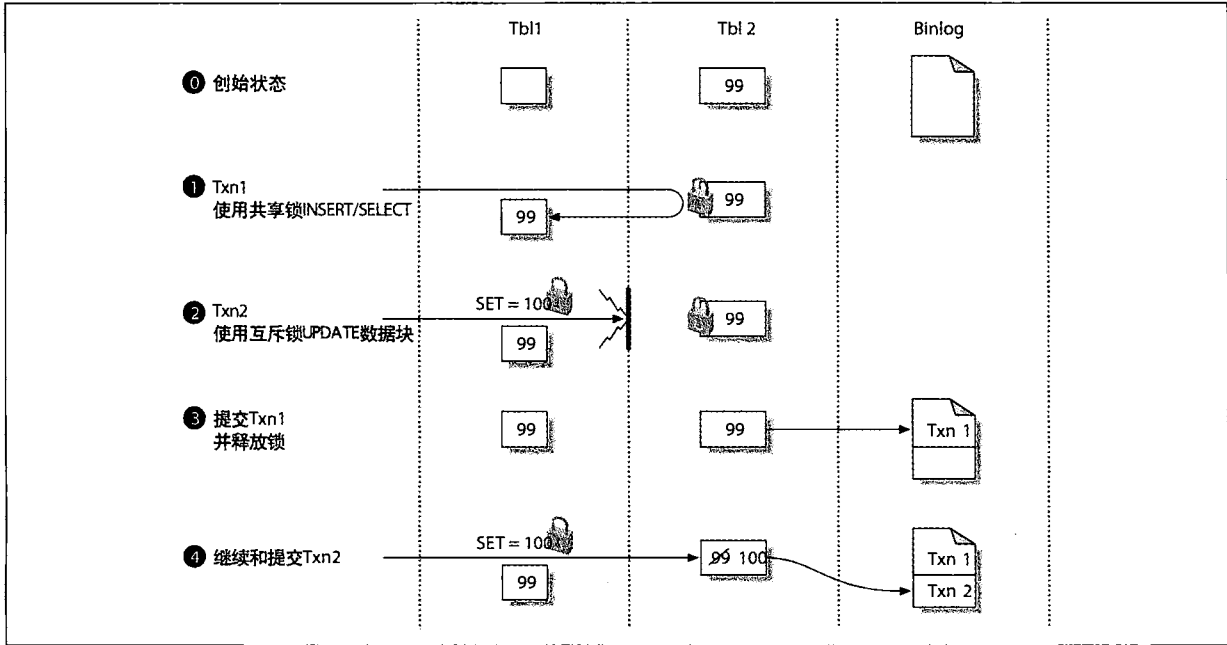


图 8-16：两个事务使用共享锁串行更新数据

397

该顺序中的步骤 2 非常重要。在此步骤中，事务 2 试图更新原始表，它要求在要更新的行上防止互斥锁。互斥锁和其他锁都不兼容，包括事务 1 放置的共享锁，所以事务 2 必须等待事务 1 提交。二进制日志中事务就按照提交的顺序被串行了起来，所以重放会有同样的结果。

从另一方面来说，如果事务 1 没有在读取的行上放置共享锁，这种保证就不复存在了。研究一下图 8-17，它显示了没有锁时一种可能的顺序。

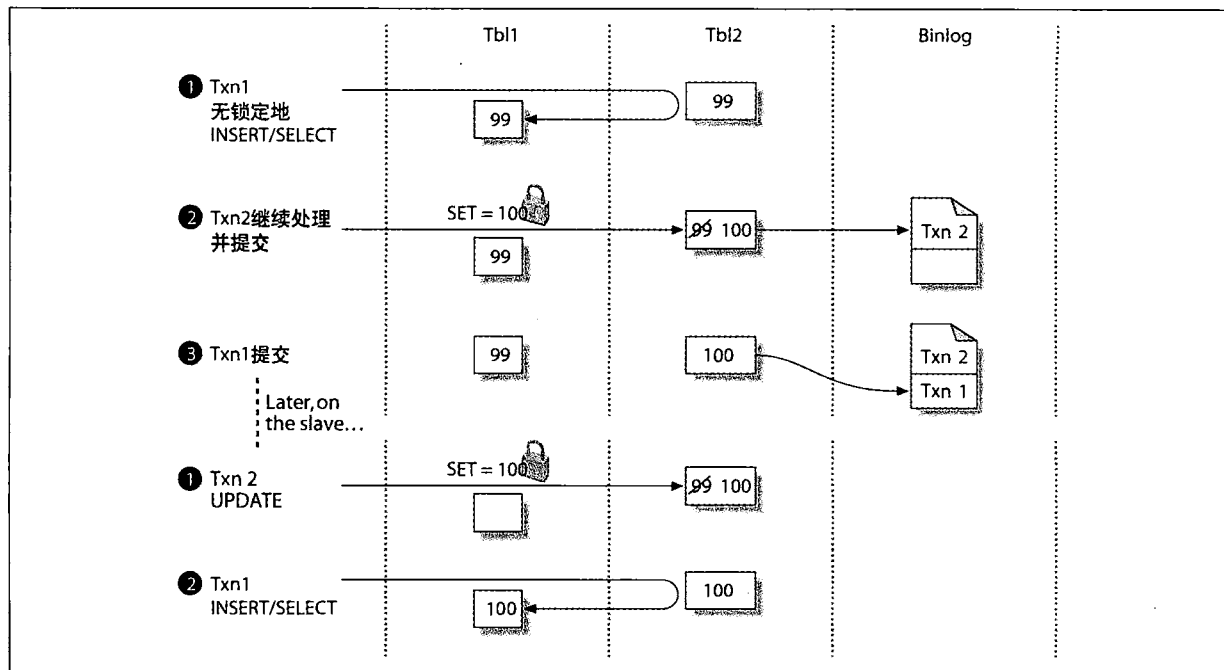


图 8-17：两个事务更新数据，但是没有共享锁串行更新

由于缺乏锁定，记录到日志中的顺序可能会在重放的时候导致不同的结果。MySQL 首先记录了事务 2，因此它会影响从服务器中事务 1 的结果。这不会在主服务器上发生，这样主从服务器的数据就不一致了。

在大多数情况下，我们强烈建议把 `innodb_locks_unsafe_for_binlog` 的值保留为 0。

8.7.13 在主—主复制中写入两个主服务器

Writing to Both Masters in Master-Master Replication

向两个主服务器写入通常不是一个好主意。如果你正在尝试同时向两个主服务器安全地写入，一些问题有解决办法，但是并非所有的问题都有解决方案。

在 MySQL 5.0 中，两个配置变量有助于解决 `AUTO_INCREMENT` 主键冲突的问题。它们是 `auto_increment_increment` 和 `auto_increment_offset`。可以使用它们来对服务器产生的数据进行“分层”，这样它们就是交错的，而不会相互冲突。

然而，这不能解决向两个服务器写入的所有问题。它只解决了自增的问题，这只是问题的一小部分。事实上，它增加了一些新问题：

这是一个警告信号！它有可能意味着从服务器负担太重，在负载最高的时候无法跟上主服务器。一种粗略的估计有多接近于其容量的方式是停止从服务器的 SQL 线程一段时间，然后重新启动，看需要多长的时间才能再次跟上。

Google 已经发布的补丁中（参阅第 451 页“同步的 MySQL 复制”）还包含了一个 `SHOW USER STATISTICS` 命令，它可以显示复制用户的 `Busy_time`。它表示了从服务器线程用来处理查询的时间的百分比，是一种了解从服务器 SQL 线程还有多少空间的好办法。

在从服务器上记录查询，并且使用日志分析工具来参看到底什么东西慢是最好的方式。不要依赖于自己的直觉，也不要基于查询在主服务器上的性能进行判断，因为主、从服务器的性能特点很不一样。最好的分析方式是在从服务器上开启一段时间的慢速查询日志。标准 MySQL 慢速查询日志不会记录从服务器线程执行的慢速查询，因此不能在复制的时候看到哪个查询缓慢。微秒级的慢速查询补丁解决了这个问题。更多关于慢速查询日志和补丁的信息请参阅第 63 页“MySQL 分析”。

除了购买更快的磁盘和 CPU 之外，从服务器其实没有太多的调优空间。大部分选择都是禁止一些引起额外工作的选项，以减轻负载。一种容易的改变就是配置 InnoDB，使其不要频繁地把改变刷写到磁盘上，因此事务提交会更快一些。可以把 `innodb_flush_log_at_trx_commit` 设置为 2 达到这一目的。还可以在从服务器上禁止二进制日志，把 `innodb_locks_unsafe_for_binlog` 设置为 1，并且把 MyISAM 的 `delay_key_write` 设置为 ALL。但是这些设置是以安全性换取速度。如果把从服务器提升为主服务器，要确保把这些值重新设置为安全值。

不要重复写入中昂贵的部分

重新架构应用程序，并且/或者优化查询通常是帮助从服务器跟上的最佳方式。要尝试最小化系统中的重复工作，任何在主服务器上昂贵的写入都会在每个从服务器上重放。如果可以把工作移到从服务器上，那么就只有一个从服务器会进行这项工作，然后可以把结果回传到主服务器上，例如使用 `LOAD DATA INFILE`。

下面有一个例子。假设有一个很大的表，然后汇总到一个较小的表中以进行常用的操作：

```
mysql> REPLACE INTO main_db.summary_table (col1, col2, ...)
-> SELECT col1, sum(col2, ...)
-> FROM main_db.enormous_table GROUP BY col1;
```

如果在主服务器上执行这个操作，每个从服务器都会不得不重复大量的 `GROUP BY` 查询。如果进行很多这样的操作，从服务器就无法跟上。把该项工作移到一台从服务器会有帮助。在从服务器上，或许是一个特别保留的数据库，以避免和复制的数据产生冲突。可以运行下面的查询：

```
mysql> REPLACE INTO summary_db.summary_table (col1, col2, ...)
-> SELECT col1, sum(col2, ...)
-> FROM main_db.enormous_table GROUP BY col1;
```

现在可以使用 `SELECT INTO OUTFILE`，然后在主服务器上使用 `LOAD DATA INFILE` 把结果送回主服务器。现在重复的工作就被减少为简单的 `LOAD DATA INFILE`。如果有 N 个从服务器，那么就节约了 $N-1$ 个 `GROUP BY` 操作。

这种策略的问题就是处理陈旧的数据。有时难以通过读取从服务器和写入主服务器得到一致的结果（下一章将会解决的问题）。如果难以在从服务器上进行读取，你还是能简化并节省从服务器的很多工作。如果分离查询的 `REPLACE` 和 `SELECT` 部分，就可以把结果提取到应用程序中然后重新插入主服务器中。首先，在主服务器上执

行下面的查询：

```
mysql> SELECT col1, sum(col2, ...) FROM main_db.enormous_table GROUP BY col1;
```

然后可以对结果集中的每一行重复下面的查询，把结果重新插入到汇总表中：

```
mysql> REPLACE INTO main_db.summary_table (col1, col2, ...) VALUES (?, ?, ...);
```

值得再次说明的是，你已经在从服务器上节省了 GROUP BY 工作，分离 SELECT 和 REPLACE，意味着查询的 SELECT 部分不会在每一台从服务器上重放。

这种通用的策略——节省从服务器的昂贵操作——可以在很多情况下提供帮助，因为查询的结果计算很昂贵，但是一旦被计算出来之后，处理就很容易。

402

在复制之外并行地写入

另外一种避免过大延迟的策略就是避免复制。任何在主服务器上执行的写入都必须被序列化到从服务器上，因此有理由认为“序列化写入”不能充分利用资源。所有的写入都需要从主服务器传递到从服务器吗？如何把从服务器有限的串行写入容量留给真正需要通过复制进行的写入？

这种考虑有助于对写入指定优先级。在特定情况下，如果可以确定某些写入可以轻易地在复制之外完成，那么就可以并行化写入，以释放从服务器的写入容量。

一个很好的例子是数据归档。OLTP 归档查询通常是简单的单行操作。如果仅仅把不需要的行从一个表移到另外一个表，那么就没有理由把这些操作复制到从服务器上。相反地，可以对归档命令禁止二进制日志，然后在主、从服务器上运行同样但分离的归档过程。

自己拷贝数据，而不是通过复制，听起来很疯狂，但是它对于某些应用程序是明智的。如果应用程序是更新某些表的唯一来源，这种想法尤其有道理。复制瓶颈通常集中在一小列表上，如果能够在复制之外处理这些表，那么就能显著地加速这一过程。

为从服务器线程准备好缓存

如果有正确的工作负载，那么就能在从服务器上通过并行 I/O 把数据预先提取到内存中而得益。这种技巧并不广为人知，但是我们知道一些大型的应用程序可以从中得益。

这个主意是安排一个程序，它能比从服务器的 SQL 线程稍微提前一点在中继日志中读取到查询语句，并将其作为 SELECT 命令来执行。这导致服务器把一些数据从磁盘提取到内存，因此当从服务器的 SQL 线程从中继日志中执行命令的时候，它就不需要等待从磁盘读取数据。实际上，SELECT 并行了从服务器必须串行处理的 I/O。当一个命令正在改变数据的时候，第二个命令的数据也正在被从磁盘提取到内存中。

程序应该在 SQL 线程前多久执行没有定值。可以尝试几秒钟，或者中继日志中相同的字节数。如果提前太多，提取到缓存中的数据就会被清空。

让我们看一个例子，了解如何利用这种方法重写命令。考虑下面的查询：

```
mysql> UPDATE sakila.film SET rental_duration=4 WHERE film_id=123;
```

下面的 SELECT 语句取得了同样的行和列：

403

```
mysql> SELECT rental_duration FROM sakila.film WHERE film_id=123;
```

只有拥有了正确的工作负载和硬件配置才能使用这种技巧。下面是一些它能工作的条件：

- 从服务器 SQL 线程是 I/O 密集型的，但是从服务器从整体上来说不是 I/O 密集型的。完全 I/O 密集型的服务器不会从预读取中获益，因为它没有任何空闲的磁盘来进行读取。
- 工作集远大于内存（这是 SQL 线程花费大量时间等待 I/O 的原因）。如果工作集和内存大小匹配，预读取不会有帮助，因为服务器将会很快“热身”好，并且很少出现 I/O 等待。
- 从服务器有许多磁盘驱动器。我们知道的使用这种策略的人，他们的每台从服务器至少有 8 个磁盘。磁盘更少的时候它也许能工作，但是至少需要 2 个或 4 个磁盘。
- 使用了有行级锁的存储引擎，比如 InnoDB。在 MyISAM 表上尝试这种办法也许会导致主、从服务器竞争表锁，甚至会使速度更慢。（然而，如果有很多表，并且写入分布在这些表上，理论上可以在 MyISAM 表上使用这种办法加快速度。）

能从预读取获益的一个例子就是广泛分布的单行 UPDATE 命令，它通常在主服务器上引起并发。DELETE 命令也可能有帮助。INSERT 命令不太可能从这种方式得益——尤其是行被顺序插入——因为前一次插入使索引已经被“加热”了。

如果表有很多索引，那么就不太可能预提取将被修改的数据。UPDATE 语句也许会修改每个索引，但是 SELECT 通常只会读取主键，在最好的情况下，会读取一个第二索引；UPDATE 还需要提取其他的索引，这降低了这种策略在有很多索引的表上的有效性。

可以使用 iostat 来检查是否是空闲的硬盘空间来处理预读取请求，检查 queue size 和 service time（示例请参阅前一章）。小队列说明一些高层因素正在串行化请求；大型队列说明高负载——这种类型在有许多磁盘的时候不容易从 SQL 线程得到。如果 service time 较大，意味着大多查询被一次性地提交给了磁盘。

404 这种技术不是“银弹”。有很多原因导致它不能工作，或者引起更多的问题，只能在清楚地了解了硬件和操作系统的环境下才能尝试这种方式。我们知道某些人利用这种办法把复制速度提高了 300% 到 400%，我们自己也尝试过，却发现它并不是一直有效。正确地设定参数是重要的，但是并没有始终正确的参数组合，有时文件系统或内核行为会使并行 I/O 失败。

Mk-slave-prefetch 工具是 Maatkit 的一部分，它是本节的想法的一个实现。

8.7.15 来自主服务器的过于巨大的包

Over-sized Packets from the Master

复制中另外一个难以追踪的问题会在主服务器的 max_allowed_packet 大小和从服务器的不匹配时被引发。在这种情况下，主服务器能记录从服务器认为过大的包，并且当从服务器取得该巨大事件的时候，就会出现各种问题。这包括错误和重试的无限循环，或者破坏中继日志。

8.7.16 受限制的复制带宽

Limited Replication Bandwidth

如果通过受限带宽进行复制，可以在从服务器上激活 slave_compressed_protocol（在 MySQL 4.0 及以上版

本可用)。当从服务器连接到主服务器的时候，它就会请求压缩连接——任何 MySQL 客户端都可以使用同样的压缩。压缩引擎使用的是 zlib，并且我们的测试表明它可以把一些文本数据压缩到原始大小的 1/3。其代价就是需要额外的 CPU 时间，在主服务器上压缩数据，在从服务器上解压缩数据。

如果主服务器和从服务器之间是慢速连接，那么就应该把分发主服务器和从服务器放在同一地点。这样就只有一台服务器通过慢速连接和主服务器相连，不仅减少了带宽的负担，也减少了主服务器 CPU 的负担。

8.7.17 没有磁盘空间

no disk space

复制确实能用二进制日志、中继日志或临时文件把磁盘塞满，如果在主服务器上执行大量的 LOAD DATA INFILE 查询，并且从服务器上开启了 log_slave_updates 选项，尤其会发生这种现象。从服务器越落后，接收到但是没有执行的中继日志就越可能占用更多的磁盘空间。可以通过监测磁盘使用并且设置 relay_log_space 来避免这个错误。

8.7.18 复制局限

Replication Limitations

复制有可能会失败。不管是否出现了错误，都有可能因为它内部局限导致数据不同步。大量的 SQL 函数和编程实践不能可靠地被复制（本章已经描述了很多这样的情况）。难以确保产品代码中不会出现这些问题，尤其是程序或团队很大的时候（注 14）。

另外一个问题就是服务器中的缺陷。我们不想显得很消极，但是大部分 MySQL 版本在复制方面都有缺陷，尤其是第一个主要版本。新特性，例如存储过程，通常会引起更多的问题。

对于大部分用户来说，没有理由不使用新特性。它只是仔细地进行测试的一个理由，尤其是升级应用程序或 MySQL 的时候。监测也很重要，你需要知道何时会有问题。

MySQL 复制是复杂的，如果应用程序越复杂，你就应该越小心。如果学会如何使用它，它就能很好地工作。

8.8 复制有多快

How fast is Replication?

常见的问题是复制到底有多快。简单来说，它通常会很快，并且它的速度和 MySQL 拷贝与重放事件的速度一样快。如果网络连接很慢，并且二进制日志事件很大，二进制日志记录和在从服务器上的执行之间的延迟就是可察觉的。如果查询运行时间较长并且网络很快，那么在从服务器上的执行时间就会比复制需要的时间更长。

一个更完整的解释需要测量复制过程中的每一个步骤，并且决定哪个步骤需要最长的时间。一些读者仅仅在意主服务器上记录事件和把事件拷贝到从服务器中继日志之间的时间间隔。对于想了解更多细节的读者，我们可以做一个快速实验。

注 14: MySQL 没有 forbid_operation_unsafe_for_replication 选项。

我们在本书的第一版详细描述了复制过程和 Giuseppe Maxia (注 15) 使用的高精度方法。我们创建了一个非确定性的 UDF，它以毫秒精度返回系统时间。(源代码请参阅第 230 页“用户定义函数”):

406

```
mysql> SELECT NOW_USEC( )
+-----+
| NOW_USEC( ) |
+-----+
| 2007-10-23 10:41:10.743917 |
+-----+
```

这使我们可以主服务器的表中插入 NOW_USEC() 的值，然后和从服务器的值进行比较，以测量复制的速度。

为了测量延迟，我们在同一个服务器上创建了两个 MySQL 实例，以避免由于时钟引起的不精确。我们把一个实例配置为另外一个实例的从服务器，然后在主服务器实例上面运行下面的查询：

```
mysql> CREATE TABLE test.lag_test(
->   id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
->   now_usec VARCHAR(26) NOT NULL
-> );
mysql> INSERT INTO test.lag_test(now_usec) VALUES( NOW_USEC( ) );
```

我们使用了 VARCHAR 列，因为 MySQL 的内建时间类型不能以小于秒的精度存储时间（尽管一些时间函数可以以小于秒的精度进行计算）。现在剩下的就是比较主、从服务器的差异。联合表（Federated Table）很容易做到这一点。在从服务器上，运行下面的查询：

```
mysql> CREATE TABLE test.master_val (
->   id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
->   now_usec VARCHAR(26) NOT NULL
-> ) ENGINE=FEDERATED
-> CONNECTION='mysql://user:pass@127.0.0.1/test/lag_test';
```

一个简单的连接和 TIMESTAMPDIF() 函数以毫秒精度显示了主、从服务器上执行查询的时间差别：

```
mysql> SELECT m.id, TIMESTAMPDIF(FRAC_SECOND, m.now_usec, s.now_usec) AS usec_lag
-> FROM test.lag_test as s
->   INNER JOIN test.master_val AS m USING(id);
+-----+
| id | usec_lag |
+-----+
| 1 | 476 |
+-----+
```

我们使用 Perl 脚本在主服务器中插入 1 000 行，每个插入之间有 10 毫秒的延迟，以避免主、从服务器实例竞争 CPU 时间。然后创建一个临时表以包含每个事件的延迟：

```
mysql> CREATE TABLE test.lag AS
-> SELECT TIMESTAMPDIF(FRAC_SECOND, m.now_usec, s.now_usec) AS lag
-> FROM test.master_val AS m
->   INNER JOIN test.lag_test as s USING(id);
```

接下来，按照延迟时间进行分组，这样就可以看到最常见的延迟时间是：

407

```
mysql> SELECT ROUND(lag / 1000000.0, 4) * 1000 AS msec_lag, COUNT(*)
-> FROM lag
-> GROUP BY msec_lag
-> ORDER BY msec_lag;
```

注 15: 查看 <http://datacharmer.blogspot.com/2006/04/measuring-replication-speed.html>。

| msec_lag | COUNT(*) |
|----------|----------|
| 0.1000 | 392 |
| 0.2000 | 468 |
| 0.3000 | 75 |
| 0.4000 | 32 |
| 0.5000 | 15 |
| 0.6000 | 9 |
| 0.7000 | 2 |
| 1.3000 | 2 |
| 1.4000 | 1 |
| 1.8000 | 1 |
| 4.6000 | 1 |
| 6.6000 | 1 |
| 24.3000 | 1 |

结果显示了大部分小查询复制的时间都小于 0.3 毫秒，这个时间包括了主服务器和从服务器上的执行时间。

复制中没有测量的部分是事件被记录到主服务器的日志之后到达从服务器的时间。知道它是有好处的，因为从服务器越快得到事件越好。如果从服务器已经收到了事件，在主服务器崩溃的时候它就能提供一份拷贝。

尽管测量结果没有精确地显示该过程需要的时间，但是从理论上，它应该非常快（例如，只是受限于网络速度）。MySQL 二进制日志转储过程不会为事件查询主服务器，它低效而缓慢。相反地，主服务器会把事件通知给从服务器。从主服务器读取二进制日志事件是阻塞性的网络调用，它主服务器记录事件之后马上就开始发送事件。因此，有理由说只要从服务器线程被唤醒，并且网络可以传输数据，那么事件就会很快到达从服务器。

8.9 MySQL 复制的未来

The Future of MySQL Replication

MySQL 复制有很多缺点，MySQL AB 计划解决它们。第三方已经提供了一些特性和修复补丁。例如，Google 已经发布了一系列定制补丁，增加了半同步复制和许多其他的特性。（更多内容请参阅第 451 页“同步 MySQL 复制”）。

另外一种可能的附加功能是多主服务器复制，例如单个从服务器和多个主服务器。这可能是一个很难解决的问题，它还需要冲突检测和分辨能力。MySQL 5.1 的基于行复制朝这个方面迈进了一步。基于行复制以后也有可能从服务器上对事件使用多个线程，这会减轻单线程的瓶颈。

另外还有整合复制和备份 API 的计划，并且允许 MySQL 服务器自动地把自己配置为另外服务器的从服务器。

现在的 MySQL 缺乏数据一致性和正确性保证。从 MySQL 网站上的投票来看，一个最重要的需求就是在线的一致性检查，以显示主、从服务器的数据是否一致。MySQL AB 为这个任务创建了一份开放的工作日志，提供了一些关于它的基本描述。许多人也要求增强二进制日志的格式以确保数据损坏可以被检测到，MySQL AB 把这个当成了一项重要的任务。

这些和其他的改进会使 MySQL 复制更加强大和可靠。几年之后再回头看看本书，了解这段时间发生了什么改变将会是一件有意思的事情。但是，值得注意的是本书第一版预测的大部分特性都没有被实现，有一些只是被部分实现了，例如，使故障安全的复制存在于 MySQL 的代码库中，但这是一个被放弃的项目。

伸缩性与高可用性

Scaling and High Availability

本章将为您讲述如何构建一个 MySQL 架构——当它扩展到很大规模时仍然能保持快速、可靠的特点。

许多伸缩性的问题是没有事前警报的，它们总在某天突然冒出来的。如果你没有为应用制定一个伸缩性的计划，那将来可能要为它保证响应速度而大费周折。那些在业务上无法伸缩自如的公司往往会彻底垮掉，如同那句有点讽刺但却真实的俗语所说：太多的成功会毁掉你的生意。

你应该也能确信应用在各种环境下运行都不会掉链子，能够应付各种情况，比如常见的软硬件故障。它们应该把这些意外情况当作惯常的事情，更完美的是，还能自动地采取相应的应对办法。

对伸缩性和高可用性的要求经常是相伴相随的。当一个应用的规模还小的时候，高可用性这一要求并不重要，这是因为：它往往只在一台服务器上运行，而单独一台服务器出故障的可能性比较小；它使用范围比较狭小，所以“罢工”时也不会带来太多经济损失；它的用户数比较少，因此人们对其“罢工”也比较能容忍。但是，在服务器数目乘以 10 之后，其中一台服务器宕机的概率也就提高了 10 倍，同时，应用也有了更多用户，他们对其寄予了更高的期望。

如果你选择了正确的架构，然后很好地实现它，MySQL 就能获得良好的伸缩性，并也能保证它的高可用性。在本章中，我们把这些概念尽可能地分解开作为一个独立的个体来讲。首先会对这些术语作一个概述，然后把伸缩性和高可用性作为其中两个主要方面来讲解（同时也会谈及一下负载均衡）。在每部分的开头，我们都会以一段需求为开始，因为成功运作一个大型应用的关键在于尽快定义你的主要业务需求，这些需求将对应用的设计和架构有着巨大的影响力。之后，我们会从优缺点两方面来讨论对于此需求可能采用的技术和解决方案。

9.1 术语

Terminology

第一步是要明确概念。人们在非正式的交谈中往往将“伸缩性”和“性能”混作一谈，其实，两者有着非常明显的不同。我们对本章中用到的这些关键性的专用语作了如下的定义：

性能

应用满足某一目标的能力，比如预计的响应时间、吞吐量水平，以及其他在第 2 章里谈到的度量指标。

负载能力

应用能够处理的总负载，在本节接下来的内容里，我们将对“负载”的含义作进一步讨论。

伸缩性

应用通过各种方式（比如增加服务器）增大时，它能够继续保持原来性能的能力。当我们在更广泛的意义上讲到性能时，它的意思就是负载量和伸缩性两者的合一。

可用性

应用响应请求时间的百分比，经常用“几个9”来表示，比如“5个9”就意味着该应用在99.999%时间里是可用的，大致也可以说是在一年里的宕机时间为5分钟（对大多数应用而言，这是很高的可用性了）。

容错性

应用和整个系统从容地处理错误故障的能力。一个系统即使被设计成具有很高可用性，也有可能宕机。当它宕机时，一个具有容错性的应用能够尽可能多地提供系统的原有功能，好过彻底的不可用。

伸缩性是个很难解释清楚的概念。这里就用类比来说明：

- 性能就是车子的时速。
- 负载能力就是高速公路的最高限速，以及公路上的车道数。
- 伸缩性就是在不减缓交通流量的情况下，你能增加的车子数量和高速公路数量的度量。
- 可用性就是一条高速公路或一条车道可供行驶的利用率。

在这个类比中，可扩展性依赖于这些因素：换道设计得是否合理、路上有多少车抛锚或发生事故、车子是否频繁地切换车道，一般说来，车子的引擎马力是不在这些因素之内的。

换句话说，**伸缩性就是在不降低性能的情况下，给应用增加负载量的能力**。这里的关键语是“可被增加的能力”。即使你的MySQL架构是可伸缩的，但应用也可能不具有伸缩性。如果你的应用难以增加负载量，那它肯定没有伸缩性了。

容错性依赖于应用里的某个组件故障时维持其中一部分功能的能力。容错性跟自我修复能力不是一回事，后者指的是应用在故障事件中恢复全部功能的能力。容错性是伸缩性的一个重要组成部分，你在设计系统时必须要对它作一些规划。如果没有给这些组件设计容错性，那它们就会很轻易挂掉，然后又引起其他组件的故障，于是一个小问题就可能导致更多的组件不能工作。容错性也要求各个组件之间要有个清晰的界线，除非你在应用设计开始之时就可以这么做，否则到了后期就很难做到了。

假如说伸缩性就是增加负载能力的能力，那么负载能力就是一个应用所承受负载的极限，伸缩性也就是应用能够处理越来越多的负载的能力。负载其实是一个复杂的概念，因为它是针对于具体应用而言的。这里就用“社交网站”为例来说明负载量的几个度量指标（这个应用对于我们讨论的概念而言是个很好的例子）。

数据量

在你应用里堆积起来的那层“薄薄”的数据是伸缩性经常要面临的挑战之一，也是当今众多Web应用要特别处置的问题，因为它们从来不能删除数据，比如社交网站从不删除旧的消息和评论。

用户数

即使每个用户本身只有一点点数据，但是当应用中累积了大量的用户后，数据量就会不成比例地更快速增长起来。更多的用户也意味着有更多的事务要处理，它同样随着用户数的增加而成比例地增加。最后，巨大的用户数量也意味着有更多的复杂查询，尤其对于那些跟用户间关联有关的查询。（用户间关联数可以用 $N*(N-1)$ 来计算，此处的 N 等于用户数。）

用户活跃度

不是所有用户的活跃度都是一样的，每个用户的活跃度也不是个常量。假如你的用户突然变得很活跃了，比如有部新片子他们都很喜欢，相互讨论很热烈，于是，你的负载就会很显著地增长起来。用户活跃度不

仅仅是页面浏览量的问题——即使是一样多的页面浏览量，如果你的网站的某个部分突然人气很旺了，也会带来更多工作量。一些用户也会比另外一些用户活跃，跟普通的用户相比，他们拥有更多的朋友，更多的消息，更多的照片。

相关数据集的大小

如果一群用户之间有着众多的关联，应用就可能会针对这整个用户群作查询和计算，这比单独一个用户要复杂得多了。社交网站经常要面对的就是那些有着众多朋友的人气很旺的用户或者群。

412 9.2 MySQL 的伸缩性

Scaling MySQL

如果把所有应用都简单地放在一个 MySQL 数据库上，那是不会有好的伸缩性的。随着负载的增加，你早晚会遇到性能方面的瓶颈。对此，传统的解决办法是购买更多更强大的服务器来替换老的服务器，这个做法就是常说的“垂直扩展”或“向上扩展”。有个与之相反的做法是把工作切分成几块，分布到各个服务器上去，这个做法叫“水平扩展”或“向外扩展”。许多应用里有些数据很少甚至不会再被访问和使用，完全可以把它们清除或归档，我们把这个叫做“回缩”——给它取这个名字仅仅是为了和其他两个操作对应。最后，有些数据库产品支持用联合方式来进行伸缩，它允许你访问远程数据就像访问本地的一样。MySQL 对此的支持比较有限。

一个理想化的伸缩性场景是这个本地的数据库能够依你需要它做到的那样，存储很多的数据，处理很多的查询，增长到很大的规模。许多人由此首先想到的是创建一个“集群”或“网格”来无缝地替代这个本地服务器，这样，应用就用不着干任何“脏活”了，也用不着知道数据到底存放在哪个服务器上。MySQL 的 NDB Cluster 技术在某种程度上支持这个方案，但是，它在许多 Web 应用上运行得并不好，因为它有着较多的制约。这就是为什么许多大型应用都构建在 MySQL 之上，却通过另外途径来达到伸缩性。我们将在本章最后部分来讨论集群方式的伸缩性。

9.2.1 规划伸缩性

Planning for Scalability

如果一个系统的伸缩性比较差，它的典型症状是应用的处理能力难以跟上负载的增加。具体表现为查询缓慢、工作量从 CPU 密集切换到 IO 密集、并发查询的竞争及延迟越来越大。究其原因，通常就是查询复杂度的增加、一部分不会被使用的数据或索引仍旧占据着内存。你可以察觉到某一类查询与其他类型之间的不同变化，比如长的或复杂的查询经常比那些小的查询更会让系统“紧张”。

如果你的应用是可伸缩的，你可以很方便地把它部署到更多的服务器上来分担负载，这样就没性能问题了。如果它是不可伸缩的，那你只能自己集中精神来搞定这个了：努力调整服务器，如此往复。这个做法治标不治本，你可以通过规划应用的伸缩性来避免这种被动局面。

伸缩性规划的最困难部分是你要预估应用要处理多大的负载。不必估算得很精确，但是，你必须按照紧要程度来评估。如果你高估了要具备的负载能力，就会在开发时浪费一些资源，如果你低估了，那将来就难以应付那部分你没估计到的负载。

413 你也需要大概评估一下实现伸缩性的日程表，即要知道底线在哪里。有些应用里，一个简单的原型系统都能很好地运行几个月，为你省出些时间去筹资用来构建更具伸缩性的架构。在另外一个应用里，你就需要让现行的

架构至少能在两年里应付负载的能力。

这里有一些问题，你可以在做伸缩性规划时问问你自己：

- 应用架构是否已经具备了完整的功能？我们建议的许多伸缩性解决方案在某些应用体系里很难实现。如果你应用中一些核心体系还没实现，那就很难看出你要怎么把他们做成具备伸缩性的。同样地，在看到这些体系怎么真正运转起来之前，你也很难为此提供一个伸缩性的解决方案。
- 期望的负载峰值是多少？你的应用需要在峰值负载的时候也能正常工作。如果首页就像是 Yahoo! News 和 Slashdot 一样，应用还能运行吗？即使应用不是这类大众化的网站，你也仍然会有个负载峰值。举例来说，你是个在线零售商，假日期间就是应用达到负载峰值的时候，尤其那些个“恶名昭著”的在线购物的旺季里（圣诞节的前几个星期），在美国，母亲节前的周末就是网上花店的峰值时间。
- 如果系统中的每一部分都要处理负载，那么如果其中一部分故障时会发生什么？比如说，使用复制从服务器（Replication Slaves）来分担你的“读取负载”，当其中一个宕机时，你的系统还能正常处理吗？还是需要关闭部分功能来保持原来的性能？在此，你可以预先准备一些闲余的负载量来缓解这种紧急情况。

9.2.2 在实现伸缩性之前“买”些时间

在一个完美的世界里，你能够做到计划先行、有足够的开发人员、有花不完的预算等。但在现实世界里，事情就复杂了。你在实现应用的伸缩性时总要做一些妥协，典型的是你常常要把系统的大改动推迟一段时间再实行。在我们开始深入到 MySQL 伸缩性的细节上之前，你要做一些实现伸缩性之前的准备工作：

性能优化

很多时候你对系统的轻微调整都能带来性能上的巨大飞跃，比如正确地建立表索引、更换为别的存储引擎。如果你正面临着性能的极限，首先要做的是开启查询日志，并分析那些运行缓慢的查询，看看哪些查询你还可以做些优化。更多内容请参阅第 64 页的“查询的日志”。

此类调整的回报总会越来越小。在修复了大多数主要问题后，通过查询优化取得的性能提升将会越来越难。每一次新的优化都是事倍功半，反而让你的应用越来越复杂。

购买性能更强的硬件设备

升级你的服务器，或者增加服务器，有时是个有效的办法。特别是对于处于软件生命周期早期的那些应用，购买更多的服务器是个好主意。与之相对应的是，尽量让应用运行在单独一台服务器上。尽管一个漂亮、优雅的设计可以实现这种可能，但是当你需要三个人，花一个月来实现这个设计时，你大概就发现这还不如多买几台服务器来得实用，尤其是在你时间不够用、开发人员不够多的情况下。

当应用比较小，或者开始就是被设计用来分布到各个硬件上时，多购置几台服务器是个行之有效的办法。对于刚上线的应用，效果就很明显，因为它比较小，或者设计比较合理。对于那些更大、更老的应用，购买更多硬件的办法可能就不怎么灵光了，或者说就很费钱了。比如说，服务器从一台增加到三台，这是小事一桩；如果服务器从 100 台增加到 300 台那就是另外一回事了——它们很烧钱的。这样的话，在现有的系统之上多花一些时间和努力来提升性能就很值得了。

9.2.3 向上扩展

Scaling Up



升级能在一段时间内产生效果，但是当你的应用变得越来越大时，这办法就不顶用了。

第一个原因是钱。无论在你的服务器上运行的是什么软件，升级是个糟糕的财务决策。市面上是有一批性价比极好的硬件设备，但是在这个范围之外，硬件设备就变得很独特很不同寻常，相应地也很贵。这就意味着你的升级有个事实上的上限。

从经济角度来看，MySQL 本身在垂直方向并不具备很好的升级条件，因为它难以有效利用多 CPU、多磁盘的硬件环境。应用对硬件设备的有效利用程度取决于你的工作量、使用的硬件和运行的操作系统。大致上说，对于目前版本的 MySQL（注 1），8 个 CPU、14 个磁盘是它的极限了。很多人在达到这个硬件限制之前问题就冒出来了。

4.1.5 即使你的主服务器能够有效利用多 CPU，但从服务器几乎不可能比主服务器拥有更强的性能。因为从服务器线程无法充分利用其多 CPU、多磁盘，所以在同样硬件环境下，主服务器能够更轻松地处理沉重的负载。

此外，你不可能无限制地升级，哪怕再强的电脑也有极限。单台服务器的应用往往是先达到它读取性能的极限，尤其是在执行复杂的读取查询时。这些查询在 MySQL 都是以单线程执行的，它们只能使用一个 CPU，所以，金钱也无法帮它们买到更高的性能。目前，最快的服务器级 CPU 也仅仅比商用级 CPU 快两倍，增加更多的 CPU 或 CPU 核也无助于提高那些缓慢查询的运行速度。当查询到的数据结果变得很大，以至于缓存也放不下它们时，服务器也就面临着存储空间的极限了。这也就是常说的现代计算机里磁盘是使用率最高，但也是访问最慢的组成部分。

应用的伸缩性也常常是个问题。由工作量导致的设计方案的选择和局限性制约着应用能否有效地利用多个硬件设备。

基于这些原因，我们建议你不要通过升级来取得伸缩性，至少不能企图无限制地升级。如果你事先知道应用会增长到很大，可以在实行一个更好的伸缩性方案前，先购买一个更强大的服务器来对付一段时间。然而，通常说来，最后还是得通过扩容来解决这个问题。这就是我们在下一节中要讲的内容。

9.2.4 向外扩展

Scaling Out

最常见且最简单的向外扩展方法是把你的数据通过集群的办法分散（Partition）到各个服务器上去，然后使用从服务器来处理读取查询。这个技术方案适用于偏重数据读取的应用。它的缺点是重复缓存，但是如果数据不大的话，这也不算是个严重的问题。这方面的内容我们在前面的章节里已经谈到过，后面还会继续讲到。

向外扩展的另外一个常见的方法是将工作量分布到各个节点上。怎么分割你的工作量是个复杂且颇有难度的决定。回想下上面提过的理想化系统，它能无限制地伸缩——这不是能用 MySQL 轻易做到的。许多大型的 MySQL 应用无法自动分解，至少无法彻底地分解。在这个篇幅里，我们就看一下各种分解的可能性，以及它们的强项和弱项。

注 1：只要你使用 64 位的操作系统和硬件，无论多大的内存都不是问题。但是其中还有些限制，只是看上去不太明显也不严重。我们已经在第 7 章中讨论过内存使用长度了。

一个节点就是你的 MySQL 架构中的一个功能单元。如果你没有做过冗余和高可用性规划，那么一个节点可能就是一台服务器。如果你正在设计一个带有容错能力的冗余系统，那一个节点通常会以下几种情况：

- 主—主双机拓扑结构，由一台主动服务器和一台被动复制从服务器组成。
- 一台主服务器，和许多台从服务器。
- 一台主动服务器，并有一个分布式数据块复制设备（Distributed Replicated Block Device, DRBD）做备用。
- 一个基于存储区域网络（Storage Area Network, SAN）的集群。

在许多案例里，所有服务器都在一个节点上就应该有相同的数据。我们倾向于在两台服务器的主动—被动节点上采用主—主双机架构。更多内容请查看本书第 365 页的“主动—被动模式下的主—主配置”。

功能拆分

功能拆分，或者说职责拆分意味着由各个节点来完成不同的功能。我们在上文中已经提到过一些方法，比如前一章讲到了针对 OLTP 和 OLAP 的工作量怎么来设计不同的服务器。相比于把单独的服务器或节点分配给不同的应用，功能拆分采取的策略往往要比它们更进一步，前者只要在服务器或节点上保留这个应用所需的特定的数据就行了。

在这里，我们多次使用到了“应用”这个词，它所指的不是一个单独的电脑程序，而是相关的一系列程序，它们中的每一个都是能被轻易地相互分离出来并独立运行的程序。举个例子来说，如果有个无需共享数据的网站，可以按照功能领域把它们分离开：门户将各个功能整合起来；通过门户能浏览网站新闻、登录论坛、寻找帮助支持，以及查阅知识库等。每一个板块的数据都可以被存放在一个独立的 MySQL 服务器里。图 9-1 描述了这个组织方式。

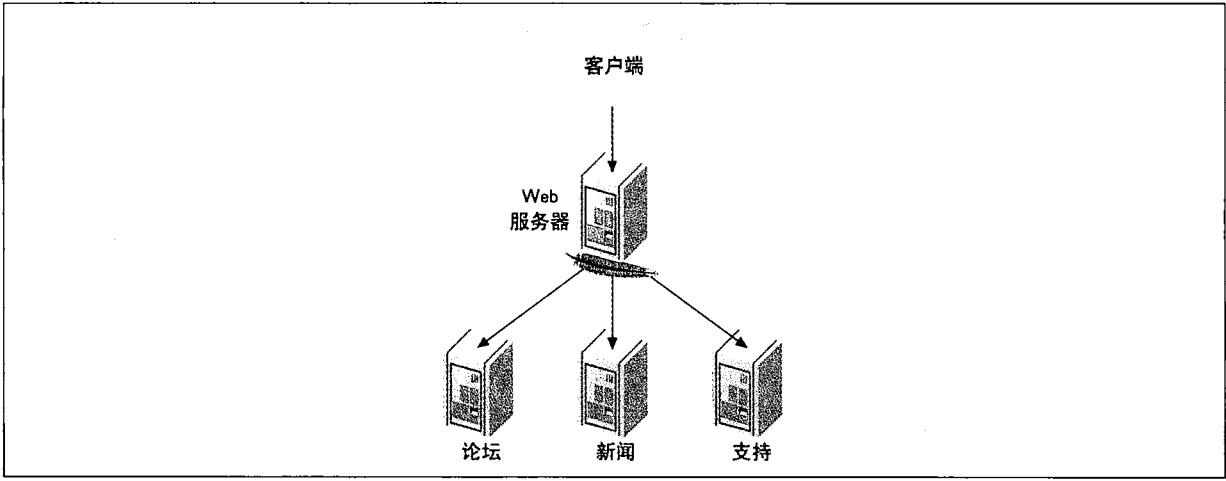


图 9-1：一个门户及 3 个专职于各功能域的点

如果应用的规模非常巨大，那每一个功能区域都可以有其专属的 Web 服务器，但是这种情形并不多见。

另一种可能的功能划分方式是把一个单独应用的整块数据按照他们关联程度，划分成几个不会相互关联的集合。当必须要关联的时候，对于那些对性能要求不高的表单，可以在应用中或者采用 Federated 表来做一些关联。这个做法在实际使用中存在着一些不同的“变种”，但它们有个共同属性，就是每一类的数据都可以在一个单独的节点上被找到。数据的划分没有一个通行的方法来做，因为这本身就很难高效地完成，任何一个方法与其他方法相比较，也不会有突出的优点。

最后，你也不可能无限制地通过功能划分来扩展，因为如果每个功能区都紧密绑定在一个 MySQL 节点上的话，就只能在垂直方向上进行扩展。假如有一个应用或一个功能区最终会变得过于庞大，那你就必须采取不同的策略了，如果你在功能分区上已经走得很远了的话，那它就更难于采用更具有扩展性的设计了。

数据分块

在目前用于扩展大型 MySQL 应用的技术中，数据分块（注 2）是一种最通用且成功的方法。把数据切成一小片，或者说一小块，然后存储在不同的 MySQL 节点上。

分块在功能分割合并时非常好用，但许多标准系统中，总有一些“全局”数据根本无法被分块（比如城市名单）。这些全局数据可以放在一个单独节点上，通常是被加载在一个像 memcached 这样的缓存里。

实际上，许多应用只将其所需要的数据分离出来使用——很显然，整个数据集的这些部分都会增长到巨大的规模。假设你正在构建一个博客服务系统，预计会有 1000 万个用户，这时也可以不把用户注册信息部门分割出来，因为你不会把所有用户（或者是其中活跃的那部分用户）的信息都放在内存里。假如用户数将达到 5 亿，那么你就可能需要数据分块了。用户产生的那些内容，如文章和评论，都需要提取出来放在同一个地方，因为它们现在已经很多了，并且还将更多。

大型应用可能有多个逻辑数据集，处理它们的方法各不相同。你可以把它们放在不同的服务器组上，但未必一定要这么做。同一份数据的分块方式有好多种，具体要根据你访问数据的方式来决定。在下文中，我们会举例说明。

418

分块技术与许多应用的设计理念有着显著的差异，这使得一个应用难以从单块数据存储更改为数据分块式的架构。如果在应用设计时就已经预计到分块，那做起来就容易多了。

许多应用在开始设计时都不是用分块来处理将来的规模扩大的情形。举例来说，你在博客服务系统中使用复制来扩展读查询，直到某一天，它不再奏效了。这时，你就把服务分成三个部分：用户信息、文章和评论，然后，放到不同的服务器上（功能分区），最后，在你的应用中把三者综合起来。图 9-2 显示的是从单一服务器演进为功能分区模式。

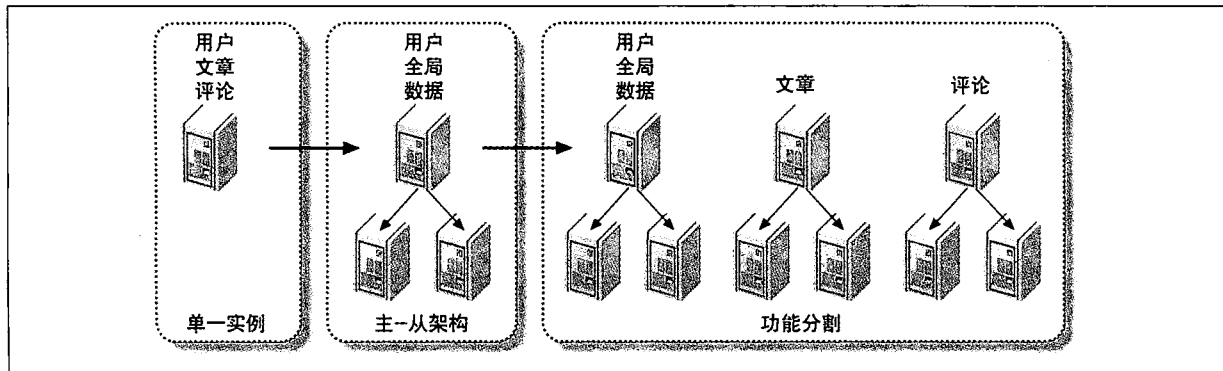


图 9-2：从单一服务器到功能分割的数据存储方式

注 2：“分块”也可以叫做“分裂”、“分割”，在这里我们就只使用“分块”来避免词义混淆。Google 一直把它叫做“分块”，如果 Google 觉得它名副其实的话，那对我们的文章而言也正合适。

最后，你还能按用户 ID 对文章和评论来进行分块，把用户信息放在一个单独的节点上。如果沿用主—从方式的配置来做全局节点，用主—主成对的方式来做那些数据分块的节点，这样的数据存储方式如图 9-3 所示。

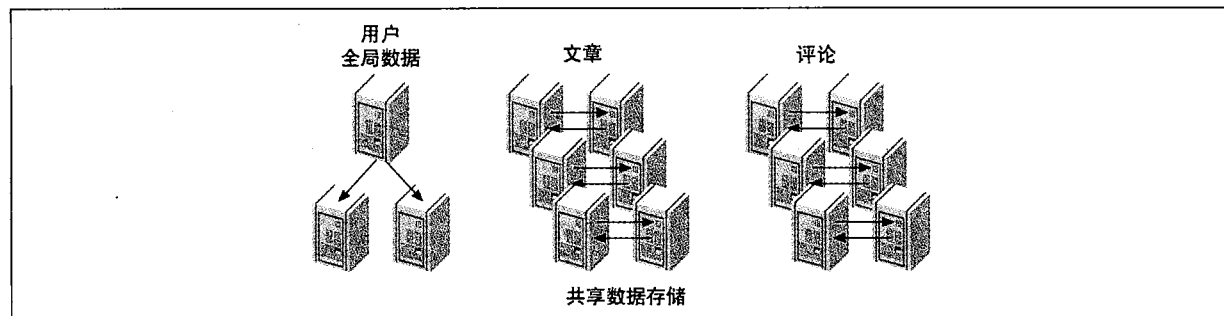


图 9-3: 1 个全局节点+6 个主—主节点的数据存储方式

如果你事先知道应用会扩展到很大的规模，而且也知道功能分区的局限性，就可以跳过这些中间步骤，直接将单一节点升级为数据分块存储方式。

采用数据分块的应用经常有一个数据访问抽象库，它降低了应用与分块数据存储之间的通信复杂度，但无法隐藏所有对分块数据存储的操作，因为应用一般都知道分块数据存储无法执行哪个查询。

太多的抽象会导致效率低下。比如在所有节点中查询一个数据，这数据其实只在一个节点上存在。这就是 MySQL 的 NDB Cluster 存储引擎在 Web 应用中表现差劲的原因之一：它隐瞒了这样一个事实，就是它查询了很多节点，但看起来像是只查询一台服务器。

数据分块存储看起来是个优雅的设计，但实际上很难构建。那为什么还要选择这个架构呢？答案很简单：如果你想扩展应用的写操作的负载能力，必须切分数据。如果只有一台单独的主机，不管从服务器有多少，你都没法扩展写操作的负载能力，对于上述的缺点而言，数据分块正是一个首选的解决方案。

如果存在着一个彻底自动的、高性能的、透明的数据分块方法，并且执行起来跟在一台服务器上一样，那真的是很棒，但是，实际上它还不存在。在将来，MySQL 的 NDB Cluster 存储引擎在这个目标方向上会运行得更快更健壮。

选择分割键

数据分块最重要的挑战是发现和获取数据。你打算怎么对数据分块就怎么去找数据。找数据的方法有很多，其中几个比其他的更好用一些。

数据分块的目标是最重要最常用的查询数据做尽量少的分块。因此，这个过程有个很重要的方面：怎样为你的数据选择一个（或多个）分割键。分割键决定了数据中的每一行将被分配到哪个数据分块中。如果你知道一个对象的分割键，就回答以下两个问题：

- 这个数据放在哪里？
- 我需要读数据时，从哪里可以获得？

在下文中，将为你展示选择和使用分割键的几种方式。而现在，让我们一起看个示例。假如我们像 MySQL NDB Cluster 一样来操作，用一张由所有表单的主键组成的散列表来对数据进行分块，并以此建立数据分块。这是个很简单的方法，但是它的扩展性很不好，因为它经常要查找所有数据分块来获取你需要的数据。比如说，你要

读取用户 3 的所有博客文章，在哪里可以找到它们呢？这些文章数据很可能被分散到了所有的数据分块中，因为它们是按数据的主键值来分割的，而不是按用户。

跨分块的查询比单一分块上的查询要糟糕，但是，只要交叉的数据分块不太多，它的情况也不是那么糟。最糟的情况是你不知道你需要数据放到哪个分块上了，那样你只能扫描每一个数据分块来查找。

一个好的分割键往往是数据库里一个很重要实体的 ID。这些 ID 可以用来区分分块单元。比如你使用用户 ID 或客户 ID 来分割数据，那么数据分块就是用户或客户。

寻找分割键的一个好办法是为你的数据模型画一个实体—关系图，或者使用相应的工具来显示数据模型里的实体和它们之间的关系。把这个关系图全部显示出来，相关的实体尽量地靠近显示。不要只是盯着图看看，而是要仔细检视这个图表，根据应用要执行的查询要求，寻找可作为分割键的字段。如果其中有两个实体之间有着关联，但是应用很少甚至从不使用这个关联，那么就可以断开这个关联以实现所需要的分块。

总有一些数据模型更易于实现数据分块，这取决于实体—关系图中那些实体的连通程度。图 9-4 里左边是一个易于实现分块的数据模型，右边那个则难以实现。

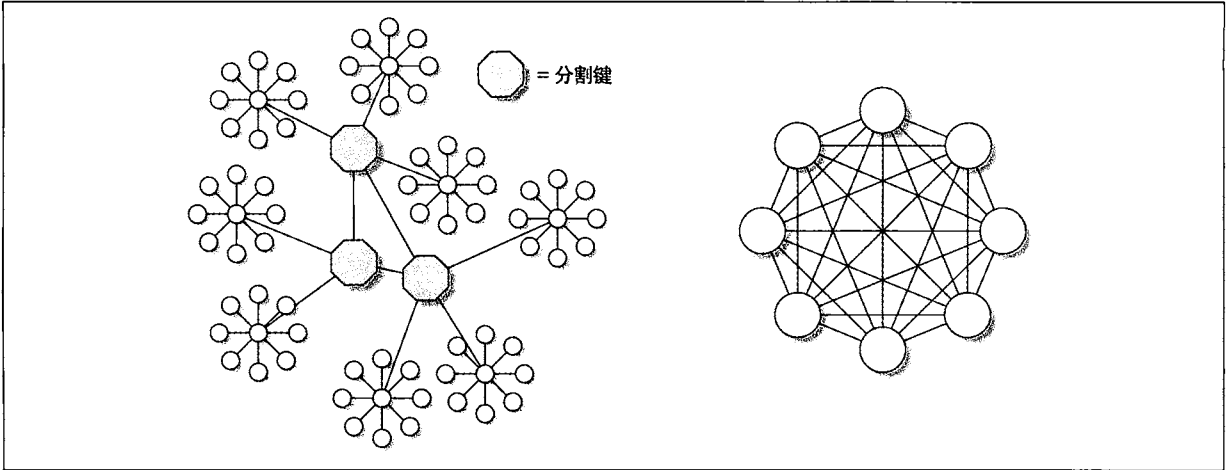


图 9-4：两个数据模型，一个易于分块，另一个则难以分块（注 3）

左边那个数据模型易于分块是因为那些子图之间只有单一连接，你可以从子图间的连接来“切割”数据。右边那个数据模型难以分块是因为它没有所谓的子图。许多数据模型都类似于左边的那个，而右边那种比较少。

多个分割键。复杂的数据模型会使数据分割变得困难。许多应用不止一个分割键，特别是当数据中存在两个或更多个“维度”的时候。换句话说，应用需要从多个角度看到一个有效的连贯的数据视图。这就意味着需要在系统里存储某些数据至少两份。

举例说，你需要把博客应用分别用两个方式分块：按用户 ID 和按文章 ID，因为这两个都是应用查找数据的常用方式。试着想象这种情形：你经常要看某个用户发布的所有文章，或者某个文章及其所有评论。如果数据只是根据用户来分块，你就难以查看某文章的所有评论；同样地，如果只是根据文章来分块，就难以查看某用户的所有文章。如果要在一个分块上同时做这两种类型的查询，就得从对应的两个方向上进行分块。

你用不着仅仅为了多个分割键而去设计一个双重冗余数据存储。让我们看一下另外一个例子：一个社交网站下

注 3：感谢 HiveDB 项目和 Britt Crawford 提供这些优雅的图表。

面的读书俱乐部站点，这个站点上的用户能够对书进行评论。站点能够显示任何一本书的所有评论，也能够显示任何一个用户所读过的书和发表过的评论。

你可以为用户数据建立一个分块数据存储，同时为书本数据建立一个分块数据存储。每一条评论既有用户 ID 又有书本 ID，这就跨越了两个数据分块的边界。无须在每个数据分块中存储一份评论数据，可以把这些评论数据都放在用户数据中，然后把每一条评论的标题和 ID 放在书本数据中。这样正好用来呈现一本书的所有评论，而不必同时访问两个数据分块，直到显示评论的全部内容时，你才需要到用户数据分块中去读取。

跨分块查询

许多基于数据分块的应用总有一些查询需要聚合或联接多个数据分块来完成。比如说，如果读书俱乐部网站要显示最受欢迎或最活跃的用户，它就必须访问每一个数据分块。要让这类查询也能很好的执行，是实现数据分块的最困难的部分。因为从应用层面来看，一条单独的查询会被拆分成多条并行执行的查询语句，每个数据分块一条。一个好的数据库抽象层可以帮你减少这种痛苦，但是这样的查询总归是缓慢的，并且比非数据分块查询昂贵得多，后者有主动缓存可供使用。

某些编程语言，例如 PHP，对于多条查询的并行执行没有很好的支持。对于这种情况，通常的做法是用 C 或 Java 写一个辅助程序，用来执行这类查询和汇集结果。这样 PHP 就可以向这个辅助程序（往往是个 Web 服务）发起查询。

跨分块查询也可以借助汇总表来进行。你遍历所有数据分块并把遍历结果作为冗余数据存放在该分块上。如果觉得同样的数据在一个分块上放两份很浪费的话，可以把汇总表放到一个另外的数据存储上，这样他们就只有一份了。非分块的数据总是被放在一个全局的节点上，并由缓存来分担其负载。

当数据的分布性极其重要，或者实在找不出合适的分割键的时候，一些应用就会采用随机分块的方式。分布式搜索应用就是个很好的例子，在这个应用里跨分段数据的查询和聚合变得很寻常，而不是偶然发生的。

跨分块的查询还不是数据分块要面对的唯一难题，保证数据的一致性也比较困难。外键在各分块之间已经不起作用了，对此的解决办法是在应用中根据需要随时检查其一致性。还有一个可能的办法是使用 XA 事务处理，但这会使经费超支，所以不常用。更多内容请查看 262 页的“分布式 (XA) 事务处理”。

你也可以设计一个间歇执行的清理进程来做数据一致性的事情。比如，如果读书俱乐部网站上一个用户的账号超期了，用不着立即把它移除，你可以写一个周期性执行的任务去删除这个用户在书本数据分块上的所有评论。也可以编写一个检查脚本，定期执行，来确保各个各数据分块之间的数据一致性。

分配数据、数据分块和节点

数据分块与节点间不一定非得是一一对应的关系。一个比较好的主意是把数据分块的大小做得比节点的容量小得多，这样，在一个节点上就可以放多个数据分块了。

数据分块的小块化有助于提高数据的可管理性，备份与恢复将更易于操作，如果表都比较小，那么像格式更改这样的操作就会更加方便。举个例子，你有一个表，其中包含了 100GB 的数据，你可以把它整个地，或者分割成 100 份，每份 1GB 的数据分块放在一个单独的节点上。现在，假设你想往表里添加索引，这样在整个 100GB 的数据分块上耗费的时间肯定是多于在 100 个 1GB 的数据分块上耗费的时间总和，因为 1GB 的数据分块刚好能全部放入到内存进行操作。还有，当你对表做 ALTER TABLE 操作时需要对数据进行冻结，冻结 1GB 的数据要比冻结全部 100GB 的数据要好很多。

小块的数据分块也更易于移动，更易于在节点中间重新分配负载能力，重新配置负载平衡。移动数据分块通常是个效率低下的过程。你往往要将这个数据分块设置为只读模式（这个功能得先在你的应用实现了），然后把数据分离出来，再把它们放入另外一个节点。在这些步骤里，我们要使用 `mysqldump` 来导出数据，然后使用 `mysql` 来重新加载它们。（如果你使用的数据库格式是 `myisam`，那么可以直接复制对应的数据文件；更多内容请参考第 11 章）。

423 除了移动数据分块之外，也需要考虑在数据分块之间移动数据的事情，最好是不要中断服务以免影响整个应用的运行。如果你的数据分块很大，那在移动一个数据分块里的全部数据时，就很难保持分块间的负载均衡，所以，需要采用另外的办法来移动其中一条单独的数据（比如一个用户的数据）。在数据分块间移动数据比移动数据分块要复杂得多，因此，尽可能不要去移动数据，这也是我们一直强调的数据分块尺寸可管理性的原因。

数据分块的相对尺寸取决于应用的需要。一个粗略的指导思想是：对我们而言的可管理尺寸的意思是数据表的尺寸应该小到能在 5 到 10 分钟内完成执行那些常规的维护指令，像 `ALTER TABLE`、`CHECK TABLE` 或 `OPTIMIZE TABLE` 等。

如果数据分块太小，那你一定会遇上了很多很多的表，太多的表会引起文件系统或 MySQL 内部结构上的问题。这方面内容可参考第 279 页的“表缓存”。过于小的数据分块也会产生大量的跨分块查询。

在节点上布置数据分块。你可能要确定如何在一个节点上部署数据分块。这里有一些常用的方法：

- 每个数据分块使用一个单独的数据库，每个数据库都是用同一个名称。当你对应用结构做镜像的时候，这个方法就很有用，在生成许多应用实例时，每个实例只会被关联到一个数据分块。
- 把表从几个数据分块放到一个数据库中，把数据分块的号码写在表的名称里（比如 `bookclub.comments_23`）。通过配置，一个单独的数据库可以支持多个数据分块。
- 每个数据分块都有一个独立的数据库，并把所有应用的表都放在数据库里。把数据分块的号码写在数据名称里（比如 `bookclub_23.comments`，`bookclub_23.users` 等）。这个方法适用于应用连接到独立的数据库时不会在其查询里指定数据库名称。这样做的好处是用不着针对每一个数据分块的查询分别做个性化处理，也更便于应用从数据分块迁移到单独数据库。
- 每个数据分块都使用独立的数据库。把数据分块的号码既写在数据库名里，又写在表的名称里（比如 `bookclub_23.comments_23`）。

如果你把数据分块的号码写在表名里了，那就需要在查询语句模板里插入一个数据分块的号码。常见的办法是在查询模板里使用神奇的通配符，像 `sprintf()` 这样的格式化函数里的 `%s`，或者用字符串插入技术。在 PHP 里可以像下面这样做：

```
$sql = "SELECT book_id, book_title FROM bookclub_%d.comments_%d... ";
$res = mysql_query(sprintf($sql, $shardno, $shardno), $conn);
```

你也可以只使用字符串插值的方法来做。

```
$sql = "SELECT book_id, book_title FROM bookclub_$shardno.comments_$shardno ...";
$res = mysql_query($sql, $conn);
```

424 在新编写的应用里这样做很容易，但是对于已经运行的应用而言，麻烦就大了。在构建新应用的时候，查询模板不是个问题，我们倾向于每个独立的数据存放一个数据分块，并把数据分块的号码既写在数据库名里又写在表名里。这样的话，执行 `ALTER TABLE` 这种操作会变得复杂，但是也有它特有的优点：

- 如果整个数据分块放在一个独立的数据库里，那使用 `mysqldump` 就可以很方便地移动它。
- 因为数据库在文件系统里是一个目录，那管理这些分块数据的文件就很容易。
- 因为数据分块间互不关联，这样就很方便查看数据分块的规模大小。
- 全局唯一的表名可以避免犯错。整个数据分块范围内的表名是唯一，当你不是在相应的数据分块上做查询时，或者把一个数据分块的数据插入到另外一个数据分块时，都会很快被发现。

现在你可能想确认下应用是否具有一些“亲数据分块”的特性。你可能受益于将一些数据分块放在一起（放在同一台服务器上、同一个网段里、同一个数据中心，或者是同一个交换网络里）去探究其中的数据访问模式。比如说，把数据按照用户来分块，然后再把用户按国家分成几块放在同一个节点上。

对已有应用做数据分块的结果往往是每个节点一个数据分块。这样的简化可以限制对应用里所有查询要做的修改的范围。分块对应用而言就是分裂性的改变，就是对应用的每一部分进行尽可能的简化。如果你在做分块后，每个节点看起来都是整个应用数据的缩微图，那就不必去修改应用里的那些查询了，也用不着费心把那些查询定向到相应的节点上去。

固定分配

给数据分块分配数据有两种方法：**固定分配**和**动态分配**。这两种方法都需要一个分配函数，它读入一行的分配键，然后返回这行数据该被放置的数据分块（注 4）。

固定分配使用的分配函数只依赖于一个分配键，散列函数和模块就是很好的例子。在这些函数里，每个输入值都通过一个分配键被映射到有限数量的存放数据的“桶”里去。

假设你有 100 个“桶”，想找出哪个“桶”来存放用户 111 的数据。如果使用取模的方法，答案就很简单：111 用 100 取模等于 11，所以你要把用户数据放在 11 号数据分块中。

如果你使用 `CRC32()` 函数的方式来做散列，那答案就是 81。

```
mysql> SELECT CRC32(111) % 100;
+-----+
| CRC32(111) % 100 |
+-----+
|                81 |
+-----+
```

固定分配的主要好处是简单、开销低，可以把它硬编码在应用里。

然而，固定分配也存在着一些缺点：

- 如果数据分块很大，并且数量不多，这样就很难在分块间做平衡。
- 固定分配使得你无法决定每一份数据放在哪个分块上，这对于那些在数据分块上没有一致负载的应用很重要。总有一些数据分块比另外一些更“活跃”，如果这些活跃的数据分块恰好是被放在同一个数据分块里，那么固定分配也无法帮你把它们中的一些转移到别的数据分块上。当你的数据都被分成一小份一小份地放在每个数据分块上时，这个问题不会出现，因为大数据量才会让一些稀有的问题暴露出来。

注 4：这里我们使用了具有数学意味的“函数”，用来指代从一个输入值（域）映射到一个输出值（范围）的过程。接下来你会看到，可以用很多方法来创建这样一个函数，包括使用数据库里的查找表。

- 更改一个数据分块常常比较困难，因为它需要重新分配已存在的数据。比如说你开始时使用的是用 10 取模的散列函数，后来随着数据分块增长，打算改用 20 取模来做，这样一来，就必须把所有数据重新进行散列，随之更新大量的数据，把数据在各个数据分块之间移动。

鉴于这些限制因素，我们更倾向于在新的应用中使用动态分配策略。如果是在对已有的应用做数据分块，那还是用固定分配比较方便，因为它操作起来简单。

我们有时也会在新项目中使用固定分配。一个实例是 BoardReader (<http://www.boardreader.com>) 论坛搜索引擎，这个论坛索引了大量的数据。我们本打算通过网站 ID 来对数据做散列的，这么说的话，所有网站的论坛都会被放置到一个数据分块上，在查找一个网上的一些论坛时候会比较快速——比如说你想找出某网站上几个排名前列的热门论坛。但是，一些网站里有数以千计的论坛，其中包含了好几百万条消息。这样的数据分块就大得难以管理了，因此，我们使用了论坛 ID 来进行散列。

426 动态分配

与固定分配相对应的是动态分配，动态分配是每个数据分块单独处理。举一个例子说，就是有着用户 ID 和分块 ID 两列的表：

```
CREATE TABLE user_to_shard (  
    user_id INT NOT NULL,  
    shard_id INT NOT NULL,  
    PRIMARY KEY (user_id)  
);
```

这个表本身就是个分配函数。给予一个分配键的值（这里就是用户 ID），你就可以找到对应的分块 ID。如果找不到，则可以匹配适合的数据分块，然后把分块 ID 加入到表里。稍后也可以修改它——这就是动态分配的含义。

动态分配在分配函数里会有额外的开销，因为它需要调用外部资源，比如目录服务器（存放映射表的数据存储节点）。这样的架构需要为了效率而多一些分层结构，比如采用分布式缓存系统把目录服务的数据保存在内存里，因为这些数据平时改动都不大。

动态分配的最大好处是对数据的存储可以做精细的控制，能够把数据在分块间均匀地分布，当做一些事先未预计到的变动时更具有弹性。

动态映射可以帮你在简单的键—分块映射之上构建多层次的分块策略。例如，你可以构建一个对偶映射，把每个数据分块关联到一个组（比如读书俱乐部里的一个用户群），然后这些组尽量都放在一个数据分块里。这样应用就可以受益于这些数据分块的机密性，避免过多的跨分块查询。

如果使用动态分配，你会发现数据分块间的不均衡。当你的服务器性能不一样时，这倒是很有用。或者其中几个分块你有特别用途的时候（像做数据归档之类的），如果你有能力让各个数据分块随时保持平衡的话，可以维持分块与节点之间一对一的映射，避免了容量的浪费。一些人还是喜欢一个节点一个数据分块这样的简单配置（但是，要记住小的数据分块好处多多）。

动态分配和灵活运用数据分块的紧密性可以帮你减轻随着应用规模扩大而带来的跨分块查询的问题。设想有一个跨分块查询涉及了 4 个节点上的数据存储，在固定分配情况下，无论怎么查询都要访问所有的分块，但使用动态分配策略的话，只要在其中的 3 个节点上做一次相同的查询就够了。这样的比较看起来差别不大，但是假如你的数据分块达到 400 个时，情况将会变得怎么样：固定分配要查询 400 个分块，而动态分配可能只需要查

询 3 个分块就行了。

动态分配会让你的分块策略想要多复杂就有多复杂，固定分配就没那么多选择了。

混合动态和固定分配。你也可以采用固定分配和动态分配混用的方法，这种方法有时很有用，而且是必须的。动态分配在目录映射的规模不大时效果不错，但如果有很多分块单元时，效果就不行了。

举例说明，有个系统是用来保存两个网站之间的链接的。这样的网站存储着几百亿条记录，它所用的分割键结合了源地址和目的地址的 URL（这两个 URL 里的任何一个都有好几百万条链接，因为两个单独的 URL 都不具备做分割键的条件）。然而，把所有源地址和目标地址 URL 都放在映射表里不太可行，因为这样的话记录实在太多，每个 URL 都要占用不少存储空间。

我们的解决方案是把 URL 通过散列放入固定数量的“桶”，然后把“桶”动态地映射到数据分块上。如果你准备了足够多的“桶”——比如说 100 万个——你就能够把大部分数据分配到每个数据分块上了。这样做的结果是你获得了动态分配的好处，同时避免了巨大的映射表。

显式分配

第 3 种分配策略是让应用在创建每一行数据时显式地指定一个所需的数据分块。这种做法在已有的数据上很难做到，因此，在对现有应用做分块改造时很少被采用。然而，它有时也能发挥其优点。

这个主意是把数据分块的号码编码到 ID 中，类似的技术我们已经在主—主复制系统中用来避免重复的键值（更多内容请查看“在主—主复制系统里往两个主服务器里写入数据”，第 398 页）。

举例说，假设你在应用要创建一个用户 3，并把他的数据放在 11 号数据分块上。可以使用一个 BIGINT 类型的字段来存放这两个编号，其中的 8 位用来保存这个数据分块的号码。这样，最终的 ID 就是 $(11 \ll 56) + 3$ ，即 792633534417207299。应用可以很方便地从这个 ID 里分解到用户 ID 和分块 ID，分解的过程像下面一样：

```
mysql> SELECT (792633534417207299 >> 56) AS shard_id,
-> 792633534417207299 & ~(11 << 56) AS user_id;
+-----+-----+
| shard_id | user_id |
+-----+-----+
|      11 |       3 |
+-----+-----+
```

现在假设你要给用户 3 添加一条评论记录，并放在同一个数据分块上。应用会先在这个评论 5 和用户 3 之间做一个关联，然后用同样的方法把评论的 ID 5 和数据分块号码 11 记录在同一个 ID 里。

这种做法的好处是每个对象的 ID 里会一直带有分割键，而其他两个方法里都需要做一次联接或查找来确定分割键。如果你想在数据库里找到某一段评论，你无须知道它是属于哪个用户的；对象 ID 就直接告诉你哪里可以找到该评论。如果这个对象是用用户 ID 动态地分块了，那就不得不先找到该评论的所有者，然后通过目录服务器找到要查找的数据分块。

还有个共同存放分割键的解决方案是把该键放在单独的一个字段里。举例来说，你不会单独去引用评论 5，但是评论 5 是属于用户 3 的。这个做法让一些人看来会很开心，因为它不违背第一范式。但是，额外的字段会引起更多的开销、编码任务、以及不便之处（反过来就是我们把两个值放入同一个字段时的好处）。

显式分配的缺点是分块数据去向都是固定的，难以做负载平衡。不过，有个很好的办法，就是让它和动态分配混合着用，原先都是把数据哈希到固定数目的“桶”里，然后再把“桶”映射到节点上，现在代之以把“桶”

作为每个对象的一部分直接编码在里面。这样应用就能控制数据的存放位置了，也就能够把相关的数据放在同一个数据分块上。

BoardReader 使用的是一个变种的技术：它把分割键编码在 Sphinx 文档 ID 里。这样每个搜索结构的相关数据都能很容易地被找到。更多关于 Sphinx 的内容可以阅读附录 C。

我们在此描述了混合的方式，只是因为它们在某些案例里很管用，但通常情况下我们并不推荐这种做法，倾向于尽可能地使用动态分配的方法，尽量少用显式分配的方法。

数据分块的重平衡

只要是必需的，你就要在各个不同的数据分块间移动数据来重新平衡负载。举例来说，许多读者可能听说过那些大型图片共享网站或热门社交网站的开发人员提到他们使用自己的工具在各个分块间移动用户数据。

在分块间移动数据的好处很明显，具体来说，当你升级硬件时，它可以帮你把用户数据从旧的分块移到新的分块上，而用不着停掉全部数据分块或把它设为只读。

然而，我们也要尽量避免去做重平衡，因为这毕竟会影响用户的使用。同时，要在应用里增加一个在分块间移动数据的功能并非易事，因为其他新功能做起来时都要包含一个重平衡的脚本了。如果你的数据分块一直保持得足够小，那你就用不着操这个心了。如果你确实经常要做重平衡，那么移动全部数据分块比移动其中的一部分容易得多（而且更高效——用每行数据开销这个术语来衡量）。

429 有一种较好的策略是把新数据随机分配到一个数据分块上。当一个数据分块满了的时候，就给它设置一个标志，告诉应用别再往这里放新的数据了。将来在这个数据分块容量扩大后，又可以把这个标志移除。

假设你安装了一个新的 MySQL 节点，上面有 100 个数据分块。开始时，你把他们的标志都设为 1，这样应用就知道它们正在准备接纳新数据。一旦他们中的每一个达到了足够的数据量（比如说有 10 000 个用户），你就把他们的标志都设为 0。之后，当因为节点负载不足而拒绝新账户注册时，又可以打开其中的一些数据分块，来接纳新用户的数据。

如果因为应用升级，或者在应用里添加了新的功能，甚至是由于你算错了负载值，结果导致每个数据分块的查询负载比预期高很多，那就只能把该节点上的部分数据分块移到新的节点上以减轻压力了。这个做法的缺点是当你移动那几个数据分块时，整个数据分块会处于只读模式或者离线状态。这就取决于你和用户是否能接受这个做法了。

生成全局唯一 ID

当你把现有系统转换为数据分块存储结构时，就经常会在不同的机器上生成全局唯一 ID。对于单一数据存储模式，可以使用 AUTO_INCREMENT 字段来取得唯一 ID。默认地，AUTO_INCREMENT 就是被设计用来在单一服务器上方便地生成唯一 ID 的。

对于在不同机器上生成全局唯一 ID 的问题，以下是几个解决途径：

使用 `auto_increment_increment` 和 `auto_increment_offset`

假定这里有两个 MySQL 数据库，它们也使用了 AUTO_INCREMENT 字段来取得唯一 ID，为了保证两个数据库同时具备唯一性，我们可以用一个初始值来做 AUTO_INCREMENT 的偏移量。具体来说，在这两个 MySQL

中，它们的自增长幅度设为 2，然后将其中一台的起点设为 1，另外一台设为 2（两个中不能出现 0）。这样的话，我们就可以保证一台数据库里的 ID 总是奇数，而另一台总是偶数，绝不会重复。这样的设置可以配置到服务器上的每一个表里。

这种方法很简单，而且也不依赖一个中心节点，因此是生成唯一性 ID 的上佳选择。现有的服务器也很容易用这个方法配置，特别是当你增加服务器或灾难恢复之后。

在全局节点上创建一个表

在一个全局数据库节点上创建一个带有 `auto_increment` 字段的表，应用就从这个表里来取得唯一性 ID。

使用 memcached

在 memcached API 中有个 `incr()` 函数，它能产生一个唯一性 ID 供使用。

批量分配编号

应用从全局节点上一次性地取得一批编号供自己使用，用完后，再申请一批。

使用复合值

你可以使用一个复合的值来做唯一性 ID，比如一个数据分块 ID 和自增长编号，如何生成这样的值可以参见前面单元里讲过的内容。

使用双字段 AUTO_INCREMENT 键

这个只能在 MyISAM 表里使用

```
mysql> CREATE TABLE inc_test(
->   a INT NOT NULL,
->   b INT NOT NULL AUTO_INCREMENT,
->   PRIMARY KEY(a, b)
-> ) ENGINE=MyISAM;
mysql> INSERT INTO inc_test(a) VALUES(1), (1), (2), (2);
mysql> SELECT * FROM inc_test;
+----+-----+
| a | b |
+----+-----+
| 1 | 1 |
| 1 | 2 |
| 2 | 1 |
| 2 | 2 |
+----+-----+
```

使用 GUID

你可以使用 `UUID()` 函数来生成全局唯一的 ID。注意，尽管这个 GUID 不能被正确的复制，但是在应用中可用它来选择数据到内存中，并作为字面行的声明。GUID 的值很大且不连续，因为它不适合做 InnoDB 表里的主键，请参考“在 InnoDB 中按主键插入数据”，第 117 页。

MySQL 的开发者已经给 MySQL 创建了一个新的 `UUID_SHORT` 函数，它能返回一个较短的连续的值，也更适于用作主键。在 MySQL 的未来版本里可能会加入这个函数，在此书正在写作的时候，这样的 MySQL 版本还没发布。

如果你使用的是一个全局分配器来产生唯一性 ID，要当心它会成为应用的瓶颈。

虽然 memcached 运行很快（每秒钟几万个值），但它不是持续不断的。每次重启了 memcached 服务，都需要在缓存里初始化那个产生值。这就要求你初始化时，每次都要找出目前在各个数据分块使用的最大值，这个过程相当缓慢而且难以自动执行。

如果你使用 MySQL 里的一个表，有种方法，即创建一个 MyISAM 表，里面只有一个字段：AUTO_INCREMENT 字段。这样外部应用也可以通过事务来读取以提高响应速度。

431 可以让表随着数据的添加而自动增长，也可以使用 REPLACE 让表里的数据值保持一条。

```
CREATE TABLE single_row (
  col1 int NOT NULL AUTO_INCREMENT,
  col2 int NOT NULL,
  PRIMARY KEY(col1),
  UNIQUE KEY(col2)
) ENGINE=MyISAM;
```

你可以像下面这样使用这个表来产生唯一性 ID：

```
mysql> REPLACE INTO single_row(col2) VALUES(1);
```

在这个语句执行后，可以使用 MySQL 的 API: mysql_insert_id() 来获得这个新生成的值。这个方法在不同语言中的实现都不太一样，以下是一个 PERL 的例子：

```
my $sth = $dbh->prepare('REPLACE INTO single_row(col2) VALUES(1)');
while ( my $item = @work_to_do ) {
  $sth->execute( );
  my $id = $dbh->{mysql_insert_id};
  # Do the work...
}
```

用不着再使用另外的查询语句（比如 SELECT LAST_INSERT_ID()）来获取这个值。额外的查询又会在服务器之间多一次往返，这显得很没效率。

数据分块的工具

在设计一个数据分块的应用时，首要的事情是编写能查询多个数据源的代码。

如果没有任何抽象，让应用直接去访问多个数据源，那绝对是个糟糕的设计，因为这会增加许多复杂的代码。一个好办法是，把这些代码都放在一个抽象层里，这个抽象层要完成以下这些任务：

- 联接到正确的数据分块上，并查询数据。
- 分布式一致性检验。
- 跨分块查询后的数据聚合。
- 跨分块连接。
- 锁和事务管理。
- 创建新数据分块（至少是在运行时找到新的数据分块），然后重平衡所有分块（如果你有时间实现它）。

你可能不需要从零开始构建数据分块结构。这里有一些工具和系统，他们或者能提供一些必须的功能，或者能用来实现数据分块的架构。从最基本的层次上讲，可以使用一个工具，比如 MySQL Proxy 来抽象多个数据源的

复杂性。随着 MySQL Proxy 的逐渐演变，它在许多数据分块应用中的位置将越来越关键。

在 Hibernate Shards (<http://shards.hibernate.org>) 里已经有了一个带有分块技术的数据库抽象层，Hibernate Shards 是开源的 Hibernate ORM 库的一个扩展，是使用 Java 编写的。Google 开发了 Hibernate Shards 中 20% 的功能，然后将代码贡献给了社区。它在 Hibernate Core 接口上实现了“分块感知”，这样一来，应用在使用分块技术的时候就无须重新设计了。事实上，应用无须知道自己是不是在使用数据分块。Hibernate Shards 能够透明地从各个数据分块上读取或存入数据，但它本身也没提供一些别的功能框架，像分块重平衡、聚合查询结果等。它使用固定分配策略将数据分配到数据分块。

另一个数据分块系统是 HiveDB (<http://www.hivedb.org>)，一个 MySQL 数据分块的开源框架，它致力于用简洁明了的方式来实现数据分块的核心思想。它有着其他系统所不具备的功能，比如创建数据分块、在分块间移动数据（就是重平衡）。HiveDB 使用的是动态分配策略，它把这样的分块方式叫“水平分割”。

Sphinx 是全文搜索引擎，不是一个数据分块存放系统，但它在一些跨分块的数据查询时非常有用。它能并行地查询多个远程系统，并将查询结果聚合起来返回——这个是一个分块存储系统难以做到的功能（更多关于 Sphinx 的内容请查看附录 C）。

9.2.5 回缩

Scaling Back

对付日益增长的数据量和系统工作负载的最简单的办法是把那些不需要的数据归档和删除。根据工作负载，你会发现归档和删除不需要的数据会有显著的成效。这种做法并不是为了替代水平扩展，但是，它可作为争取时间的短期策略，也可能成为应付数据增长的长期策略之一。

当设计归档和删除策略时，这里有一些需要考虑的事情：

对应用的影响

一个设计良好的归档策略，它能够在不影响事务处理的情况下，把数据从重负载的 OLTP 服务上移走。这里的关键是能够有效地找到要移除的行，并一小块一小块地移除。你要确定每次做归档的行的数目，使这个事务在锁竞争和事务负载量之间做到很好的平衡。

哪些需要归档

当你知道一些数据不会再被使用后，就马上可以把它们删除或归档，但是你可以设计应用，使它能自动归档那些很少被访问的数据。可以把归档的数据置于核心表的附近，通过视图来访问它们，或者干脆把它们移到别的服务器上。

深度优先还是广度优先

数据间的关联使归档和删除变得复杂。一个设计良好的归档任务能够使数据保持逻辑上的一致性，至少是在应用要访问的数据范围内保持一致性，不会往庞大的事务里再加入多个表。

当表间关系众多时，决定从哪个表开始归档往往是个挑战性任务，如果没有选对开始的表，你在归档时就不必面对“孤儿”或“寡妇”数据行，因此，在归档时，决定是否违背外键将是件麻烦事（可以通过 `FOREIGN_KEY_CHECKS=0` 来关闭 InnoDB 的外键约束），或者暂时把那些“摇摆指针”式的记录放一边去。采用哪种方式取决于应用怎样看待数据。如果应用里从上到下显示一组相关的表里的数据，那归档的次序

也要照它那样。举例来说，应用总是先检查订单然后再检查发票，那你就该先将定单数据归档，这样你就不会看到成“孤儿”的发票数据了，接下来就是将这些发票数据归档。

避免数据丢失

如果你正将一台服务器上的数据归档到另一台上，可能并没用到分布式事务处理，而是把数据归档到 MyISAM 里，或者把另外的非事务数据存储引擎里。因此，为了避免数据丢失，在删除数据源之前要先把数据存放到目标服务器上。把这些归档数据同时写进一个文件倒也是个主意。你应该把归档任务设计成这个样子：无论何时把它停止或重启，都不会引起数据一致性、索引冲突之类的错误。

解归档

你会经常使用解归档的策略来减少归档数据。即在你难以确定哪些数据需要归档时通过设置一个可回退的选项来做归档。如果你设置了一些检查点让系统来检查是否有数据需要归档——这是个相当容易实现的策略。比如说你将那些不活跃的用户归档，这个检查点应该设在登录验证过程里。如果有一个登录因为不存在这个用户而失败，你就去检查归档数据，看看是否有这个用户存在，如果存在的话，就从归档数据里读取该用户的信息，完成该登录。

Maatkit 中有这样一个工具，它能高效地帮你归档和（或）删除 MySQL 里的表，但是它不提供解归档的功能。

4.3.4 保证归档数据的隔离

即使实际上不是把老数据移到其他别的服务器上，但许多应用还是得益于将活跃数据与不活跃数据相互隔离。隔离之后，缓存的利用就更有效率，而且对于这两类数据可以采用不同的硬件和软件架构来处理。这里例举了几个做法：

将表分成几部分

把表分成几部分是个聪明的办法，尤其是当表无法全部放入内存的时候。举例来说，你可以把用户表分为活跃用户表和不活跃用户表。你可能在想这纯粹是多此一举，数据库自然会把“热门”数据放入缓存里，其实，这个是因不同的数据引擎而异的。如果使用的是 innodb，它的缓存每次只加载一页，你在一页里可以放入 100 个用户，另一方面，假设在你的用户表只有 10% 是活跃的，这样的话，缓存里的每一页有 90% 的空间被浪费了。把这个表分成上述两部分的话，就能显著提高内存的使用率了。

Falcon 存储引擎有个行级别的缓存机制，这使得缓存的使用更高效。但是，这并不意味着 Falcon 的表不能通过活动/非活动的分隔而获得性能的提升。Falcon 的缓存每次对一页做索引，当这个索引里混合了活动和非活动的数据时，还是会降低的索引的效率。因此，在 Falcon 做活动/非活动的分隔时还是很有用的。

MySQL 分区

MySQL 5.1 提供了原生的分区表技术，它能把最近的数据都放在内存里。更多关于分区技术的内容请查看第 253 页的“表的合并和分区”。

以时间为基础的数据分区

如果你的应用要频繁地取得新数据，这样的话，新数据将比旧数据更加活跃。举例来说，BLOG 服务的流量来自于最近 7 天里发布的文章和评论，多数的数据更新也来自这个时间范围，那么他们就把这些数据都放进内存里，并在磁盘上存放一份可恢复的数据副本，供数据出错时调用。其他那些数据就放在别的地方

了。

我们也见过另外一种设计，他们在两个节点上存储着每个用户的信息。新的数据就被放入一个活跃的节点，这个节点配置了巨大的内存和快速磁盘，而且数据也被优化过以支持快速访问。另外一个节点上存储的是旧数据，有着容量巨大且慢速的磁盘。这里假设应用还可能要访问这些旧数据的——这个假设对大多数应用来说都很适用，它们响应的请求里，有 90% 以上是访问那些最新的 10% 的数据。

可以使用动态分配的策略来实现这种分块方式。举例来说，应用了分块目录技术的表的定义可以像下面这样：

```
CREATE TABLE users (
  user_id          int unsigned not null,
  shard_new        int unsigned not null,
  shard_archive    int unsigned not null,
  archive_timestamp timestamp,
  PRIMARY KEY (user_id)
);
```

在这个表里，通过一个归档脚本就可以把旧数据从活动节点移到归档节点上，当一份用户数据被移到归档节点上时，就更新它的 `archive_timestamp` 字段，通过 `shard_new` 和 `shard_archive` 字段你可以获知哪个编号的分块存放了该份数据。

9.2.6 通过集群来扩展

Scaling by Clustering

集群是另一种在服务器间做分布式负载的扩展技术。“集群”这个术语在计算机领域有着多种含义，通常而言，集群由一个局域网内的几台主机组成，其表现得像一台服务器。有一种集群的变种是联盟——访问远程服务器就像在本地一样，它要创建一个虚拟服务器就像许多服务器的代理。

集群

MySQL 的 NDB Cluster 存储引擎是一个分布式、嵌入内存的、非共享的存储引擎，它还有同步复制和节点间数据自动分区技术。与其他 MySQL 存储相比，它有着完全不同的性能特点。它在特定硬件上有着上佳表现。虽然对应用而言，它存储数据有着很高性能的表现，但对于许多 Web 应用来说，它还不能算是高性能的解决方案。

NDB Cluster 适用那样的应用：数据量相对较少，并执行简单的查询。具体而言，它适用于网站的 session 存储，文件元数据的存储等。当它执行复杂的查询，包括较多的联合时，性能就变得很差，从技术角度来讲就是当任何一条查询不是单表内索引查询而是内部各点间通讯时，性能就下降了。

NDB Cluster 是个事务系统，但是不支持 MVCC，它采用了读锁的方式，而且也不做任何死锁检验。运行当中遇到死锁时，NDB 就以超时返回的形式来处理。以读锁定和超时返回机制相结合的使用方式，决定了它对于多用户交互的应用和 Web 应用而言不是个好的解决方案。

你可以实现一个变种的集群方案，它基于 MySQL 之上，或者 MySQL 的前端，或者在 MySQL 底层。有个例子可以参考 Continuent (<http://www.continuent.com>) (注 5)，它提供了同步复制负载均衡和通过一个中间件层实现 MySQL 的故障恢复。

注 5：或者是该软件作者的另外一个开源项目——Sequoia，访问 <http://sequoia.continuent.org>。

联合是另外一个含有多种意义的术语。在数据库世界里，它一般是指从一个服务器去查询另外一台服务器的数据。微软的 SQL Server 分布式视图就是这样的东西。

MySQL 通过 Federated 存储引擎对联合提供了有限的支持，该引擎类似于 NDB 集群。它擅长于简单的查找，尽管它在别的服务器上做 INSERT 操作时性能也可以被接受。它目前采用的架构决定它对 DELETE 和 UPDATE 的操作更加低效——在最糟糕的情形下，效率更加低。

Federated 引擎在使用联合和做大数据规模的 SELECT 时表现很差。举例来说，当 GROUP BY 从一个表里取得数据时，或者是当使用 mysql_store_result（注 6）从远程服务器上取数据到本地内存里时。这个缺点在应用的数据规模逐渐增大时会引起大麻烦。Federated 的表也使得复制变得更复杂，因为一个简单的 update 需要在多个服务器里执行。

9.3 负载均衡

Load Balancing

负载均衡的基本思想很简单，就是在一个服务器集中间尽可能地平均地分配工作量。实现这一目的的通常做法是在服务器之前设置一个负载均衡器（往往是一个专门的硬件设备）。这个负载均衡设备会把接入的连接路由到最空闲的可用服务器上。图 9-5 显示的是典型的用于大型网站负载均衡配置，其中一个用于 HTTP 流量的，另一个用于 MySQL 访问。

负载均衡有 5 个基本目标：

可伸缩性

如果系统设计得正确，那可以通过在节点上增加更多的服务器来提高其处理能力。当增加更多服务器时，必须平衡各服务器的实际负载。

高效性

通过控制请求的路由走向使负载均衡帮你更有效率地使用服务器资源。这点特别重要，如果服务器的处理能力各不相同，那你就要把更多的工作量路由到那些强大的服务器上。

可用性

一个灵活的负载均衡方案里选用的服务器都是能保持时刻都可用的服务器。

437 透明性

客户端无须知道是否有负载均衡的存在，也不必关心在负载均衡器后面有多少台服务器，它们的机器名分别是什么。负载均衡器在客户端看来就是一台虚拟服务器。

一致性

如果你的应用是有状态的（如数据库事务、站点 session 等），那负载均衡设备就应该能在重定向客户端请

注 6：更多关于 mysql_store_result 的内容请查看第 161 页的“MySQL 客户端/服务器 协议”。

求时不丢失有关的状态信息。这样,就能使应用不必总是需要了解自己连接的是哪台服务器。

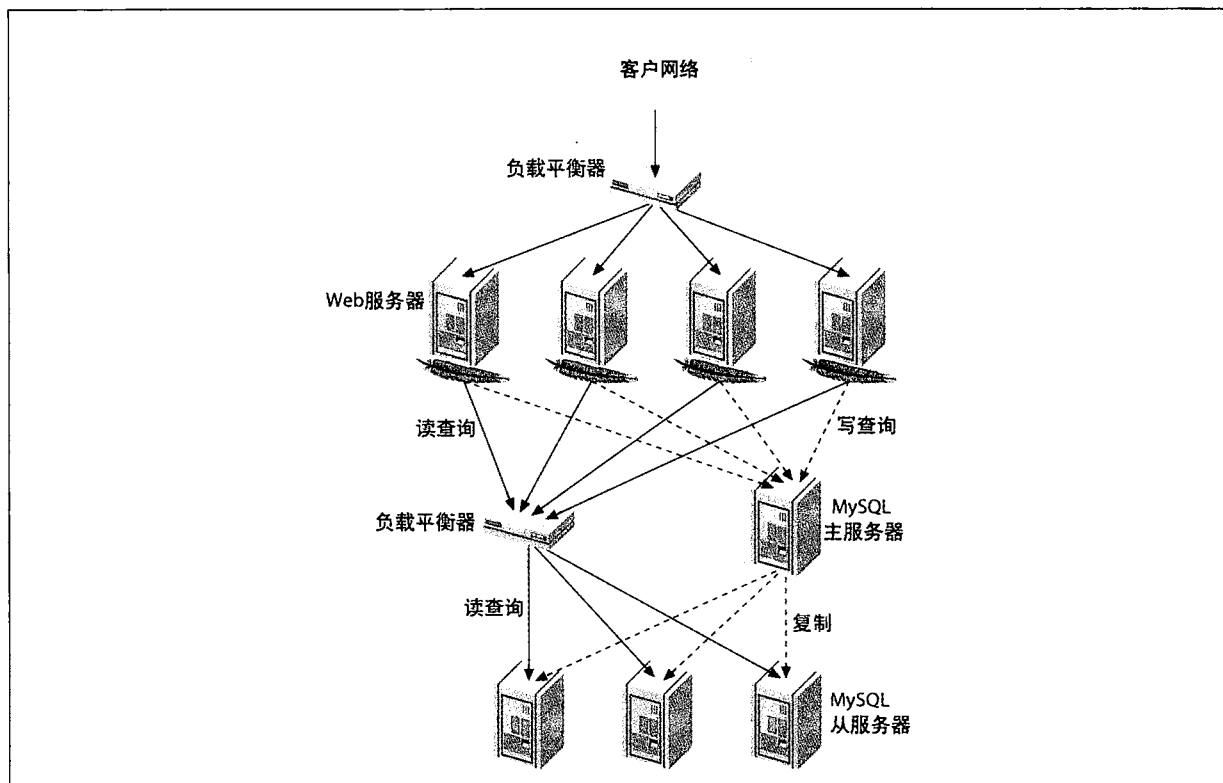


图 9-5: 典型的负载平衡架构, 用于读量较大的网站

在 MySQL 世界里, 负载平衡架构总跟数据分块和复制紧紧地伴随在一起。你可以把负载平衡和高可用性融合在一起, 部署到应用中的任何一个合适的层上。举例来说, 可以在一个 MySQL 集群的多个节点间作负载平衡, 也可以在各个数据中心间作负载平衡, 在每个数据中心里你可以使用一个数据分块的架构, 其中每个节点实际上就是一个附带了许多从存储的主—主复制对, 这些需要再次进行负载平衡。对于高可用性策略而言也是如此, 你可以在这个架构的多个层上实现故障恢复机制。

负载平衡有许多微妙之处。举例来说, 有一个挑战是管理读/写策略。有些负载平衡技术本身就实现了这个策略, 但是, 另外一些服务器就需要应用自己来记住哪些节点是可读的, 哪些节点是可写的。

当你构想如何实现负载平衡时, 以上这些因素都需要考虑进去。目前, 有很多类负载平衡解决方案供你参考, 它们的范围从以点为基本的实现 (如 Wackamole (<http://www.backhand.org/wackamole/>)) 到 Domain Name System (DNS)、LVS (Linux Virtual Server; <http://www.linuxvirtualserver.org>), 硬件负载平衡、MySQL Proxy, 以及在应用里管理负载平衡。

9.3.1 直接连接

Connecting Directly

一些人一听到负载平衡就想到那是个夹在应用与 MySQL 数据库之间的东西。然而, 这并不是负载平衡唯一的表现形式。你也可以在保持应用与 MySQL 服务器直接相连的情况下继续使用负载平衡。中央化的负载平衡—

般只有当应用访问一组对等的可替换的服务器时才有良好的效果。如果当一个应用要判断从一个从服务器上读数据是否安全时，它就需要跟服务器直接相连了。

包括一些可能出现的特例逻辑在内，应用中为负载平衡做决策的过程其实是非常高效的。举例来说，假如有两个一样的从服务器，你可以选择其中一个来应付针对某些数据分块的全部查询要求，而另外一个就应付针对其他数据分块的所有查询要求。这样就可以充分使用从服务器的内存，因为它们都只需把数据中的一部分放入内存就可以了。如果其中一台出故障了，另外一台上仍然保留了全部数据分块上的数据以供查询。

以下部分是讨论应用的“直连”技术的几个常用途径，当你评估每个选项时，应该会考虑到其中的一些注意点。

在复制上分离读与写

MySQL 复制使表有多份副本，让你选择是在主服务器还是从服务器上运行一个查询。因为复制过程是异步的，所以，这里的主要困难是怎么处理从服务器里面的旧数据。你可以把从服务器当作是只读的，而主复制能运行读、写查询。

你经常不得不对应用作一些更新，因此你得知道那些令人担心的事情（注 7）。现在，应用可以使用主服务器做读操作，而写操作可以在主服务器和其他从服务器之间分配。在从服务器里访问到的是不太紧要的旧数据，而最新数据就放在主服务器上。

439 如果你使用主—主来实现主动—被动服务器组，同样也要考虑到这些。但是，在这样的配置中，只有主动服务器来接受写操作，而读操作都让被动服务器来承担——如果它上面有那些需要的旧数据的话。

在这里的最大问题是如何避免在读旧数据产生被遗落的“史前古物”，所谓被遗落的“史前古物”是指当一个用户在对旧数据操作时，比如对博客上的某篇文章增加了一条评论后，重新加载了页面，但是看不到自己刚才提交成功的评论，究其原因应用是从一个从服务器上读出博客文章的，而新评论是写在主服务器上的。

下面罗列了一些最常用的分离读/写操作的方法：

以查询为基础的分离

最简单的分离方法是把所有的写操作和一些不接受旧数据的读操作全部放在主动服务器或主服务器上，而其他的读操作都放在从服务器或被动服务器上。这个策略很容易实现，但是在实际运行时会发现从服务器的访问没有预计的那么频繁，因为几乎很少的读操作能够只适用旧数据。

旧数据分离

这个方法是对以查询为基础分离方法的一个小小提升。这些增加的额外工作是要应用去检查从服务器的迟滞程度，然后决定这些数据对读操作是不是够“陈旧”。许多报表类的应用都可以使用这个策略：在晚上数据负荷没有的时候，就把数据复制到从服务器里，应用不会在乎这些数据是不是跟主服务器同步更新的。

以 session 为基础的分离

一个稍微更进一步的方法来决定一个读操作是否在从服务器上执行的方法是在系统标识上用户是否更改了数据。用户用不着一定要看到其他用户的最新数据，但是他必须看到自己更新过的数据。你可以这样来实现：如果用户对数据有过更新，就在 session 层设置一个标志，这样当用户执行读查询时，系统就会到主服务器上读取数据。

注 7：如果你可以使用 MySQL Proxy 来拆分查询，就不需要更改应用了。

可以把这个功能与复制间隔时间监控相结合。如果用户在 10 秒钟前更新了一些数据，并且各个从服务器在 5 秒内都更新过数据了，那么在从服务器读数据就是安全的。一个很好的主意是在从服务器中选择一台用来存储这些 session 相关的数据，否则用户会奇怪地发现有些从服务器的更新数据的速度比其他几台要慢。

以版本为基础的分离

这个方法跟以会话为基础的分离很相似：你通过版本号和（或者）时间戳来跟踪对象，根据从服务器上读取对象的版本号或时间戳来判断这些数据是否足够“新鲜”以供使用。如果从服务器上的数据太旧了，你就可以把它们从主服务器上读过来更新。即使当对象本身没有发生改变，你也可以增加顶层项目的版本号，这样就简化了旧数据的检验（你只要查看一个地方——顶层项目就行了）。举例来说，当用户在博客里发布了新文章，你就可以更新他的版本号，这样数据就会从主服务器上读取了。

在从服务器上读取对象的版本号会增加它的运行开销，只能通过缓存减轻它的影响了。在下一章里，我们会更深入地讨论缓存和对象版本。

全局版本/会话分离

这是以版本为基础和以会话为基础的变种方法。当应用执行一个写操作时，它会在事务提交后运行 `SHOW MASTER STATUS`。它把主服务器的日志坐标作为更新后的对象和（或者）会话版本号放在缓存里。然后，当应用连接到从服务器时，它就运行 `SHOW SLAVE STATUS`，用从服务器上的参数与先前保存的版本相比较。如果从服务器上的版本至少有一个编号高于主服务器提交事务后得到的版本号的话，那说明从服务器上的数据可以被安全地读取。

许多读/写分离解决方案都需要监控从服务器的滞后时间，并以此来决定从哪个服务器上读取数据。如果你也是这样做的，那请注意 `SHOW SLAVE STATUS` 执行后返回的结果里，`Seconds_behind_master` 字段不能用来衡量从服务器的滞后时间。更多细节可查看 379 页的“度量从服务器的滞后时间”。

如果纯可测量性是你的目标，你也不关心要用到多少硬件设备，那事情就变得更简单了，可以不用复制，或者只用它来提高可用性而不是做负载均衡。这样一来就避免了在主服务器与从服务器之间分离读操作的复杂性。有些人认为这么做很合理，也有些人认为这是浪费硬件。不同的目标导致了这个分歧：你是只需要可测量性，还是既要测量性也要高效性？如果你同时需要高效性，那从服务器除了保存数据副本外还要承担其他一些任务，你就不得不处理这些额外增加的复杂性了。

改变应用的配置

还有一个分担负载的方法就是配置你的应用。举例来说，在产生一批大数据量的报表时，可以多配置几台机器来分担这个负载，每台机器都连接到不同的 MySQL 从服务器，分别为对应的第 N 个客户或网站生成报表。

这样的系统通常很容易实现，但是它需要修改一些代码——包括那些配置文件会被改得诘屈聱牙。那些所有硬编码的东西因其固有的限制，需要你逐个服务器去修改，或者在一个中心服务器上修改，然后通过文件或代码控制软件的更新命令“发布”到其他服务器上。如果你的配置都是放在数据库和（或）缓存里的，那就可以避免刚才讲到的麻烦事了。



这个负载均衡的方法比较粗陋，但对于一些简单的应用来说，根据不同的目的创建 DNS 名称却很适用。你要创建一个周期性的任务来监控那些 MySQL 服务器，为每个服务器指定一个合适的名字。最简单的实现方法就是给只读的服务器一个 DNS 名字，而给负责写操作的服务器另外一个 DNS 名字。如果从服务器是与该主服务器同步的，那就把只读服务器的名字指给从服务器用，如果它们跟不上同步，你就把这个 DNS 名字给主服务器。

这种 DNS 技术很容易实现，但是错点也不少。最大的问题是你无法把 DNS 完全置于你的控制之下：

- DNS 的更改不是立即生效的，它在网络里传播到每个角落需要比较长的时间。
- DNS 数据在各处缓存起来，它的失效时间是建议性质的，而不是强制性的。
- DNS 更改后可能需要应用或服务器重启后才能生效。
- 多个 IP 地址共用一个 DNS 名称是个不好的主意，具体依赖于负载均衡请求的轮询行为，这个轮询行为不总是预计的。
- DNS 的更改不是个原子操作。
- DBA 可能无权直接访问 DNS 的设置。

除非应用非常简单，依赖于一个不可控的系统总是危险的。你可以通过另外的途径提高一点点对系统的控制权。也可以通过更改/etc/hosts 把你的控制权提升一点。当你对这个文件发布更新时，就知道这些新的设置开始生效了。这比干等着 DNS 缓存信息失效要好得多，但也不是很理想。

我们常常建议人们在构建应用时要实现对 DNS “零依赖”，即使在应用很简单时也适用，因为你不会知道应用将来会扩展到多大的规模。

移动 IP 地址

一些负载均衡的方案依赖于在服务器之间移动虚拟 IP 地址（注 8），这个方法很不错。听上去类似于改变 DNS 名称，其实不是一回事。服务器不会根据一个 DNS 名称去侦听网络流量，但是可以根据一个指定的 IP 地址去侦听网络流量。所以移动 IP 地址，可以保证 DNS 名称静止不变。通过地址转换协议（Address Resolution Protocol ARP），你能迅速而且原子性地使 IP 地址的改变通知到网络的各处。

42 有两个系统使用了这种技术：Wackamole 和 LVS。举例来说，它们让你把一个 IP 地址跟一个角色（例如只读操作）关联起来，它们根据实际需要来完成在各个机器之间移动 IP 地址。Wackamole 能够管理许多个 IP 地址，并保证有一台并且只有一台机器监听每一个地址。Wackamole 的服务是以端点为基础的，这可以用来消除一个点故障而带来的影响。

有一个方便的技术是把每个物理上的服务器都配置一个固定 IP。IP 地址就定义在服务器上，不再改变。你对每个逻辑上的“服务”使用虚拟 IP。可以为每个服务都指定一个虚拟 IP 地址，服务可以在各个服务器之间轻易地移动。这样也就可以方便地移动服务和应用实例，而不必重新配置应用了。这是个很棒的架构，你再也不必为了负载均衡和高可用性，把 IP 地址四处移动了。

注 8：虚拟 IP 地址不连接到任何一台指定的电脑或网络接口，它们只在电脑之间“漂浮”。

9.3.2 引入一个中间件

Introducing a Middleware

迄今为止，我们讨论的所有技术都是假定应用跟 MySQL 服务器是直接通信的。然而，许多负载均衡方案里都会引入一个中间件，它的作用就像个网络通信的代理，它在一边接受所有访问请求，又在另一边把这些请求指派到合适的服务器上去处理，然后又把响应结果发回到请求来源的电脑上。这个中间件有时是一个硬件设备，有时是一个软件（注 9）。图 9-6 描述的就是这个架构。这样的架构通常都会运作得很好，要是你把负载均衡设置成冗余，这样就不会因为负载均衡故障而导致整个系统瘫痪。

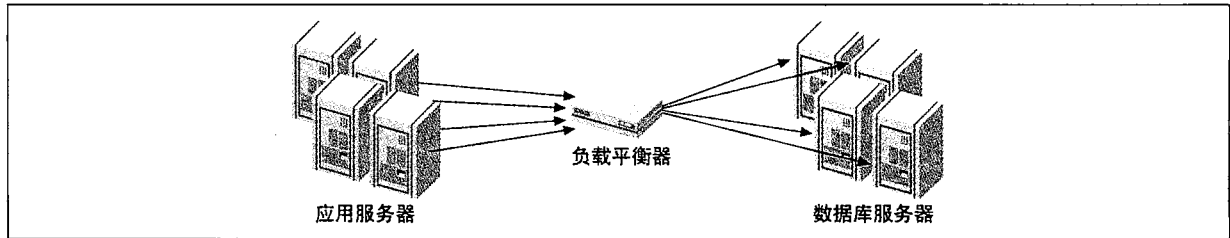


图 9-6：一个负载均衡器，其行为如同一个中间件

负载均衡器

在市场上充斥着各种各样的负载均衡设备，硬件的、软件的都有。但是，其中很少是为 MySQL 服务器专门设计的（注 10）。

Web 服务器更需要负载均衡，许多多用途的负载均衡设备都支持 HTTP，但其他基础架构却不是很全。



提示：一个例外是 MySQL Proxy，它能很好地为一些应用分离读和写操作。它增加了复杂性和一些开销。但它也带有极大的弹性，让你能通过脚本来控制读/写分离。MySQL Proxy 相对比较新，但是已经有很多在线的指南和实例来帮你配置它的负载均衡功能。因为它能深入查看由它转发的会话数据，所以，它做一些很复杂的查询路由。

MySQL 的连接都只是普通的 TCP/IP 连接，所以，可以在 MySQL 上使用多用途负载均衡器，但它们不是为 MySQL 专门设计的，所以，其中也带来了一些局限：

- 除非负载均衡器知道 MySQL 的真实负荷，否则，它不能为分布式请求做到负载均衡，不是所有查询请求都一样的，但多用途负载均衡器将它们一视同仁。
- 许多负载均衡器都知道如何检查 HTTP 请求，把会话“粘贴”到一台服务器上，并把会话状态保存在一台 Web 服务器上。MySQL 的连接也是有状态的，但是这类负载均衡器不知道怎么把所有连接请求从一个 HTTP 会话“粘贴”到一台 MySQL 服务器上。这就导致了效率的下降（如果一个会话里的所有请求都转发到同一台 MySQL 服务器上，那服务器的缓存才会显得有效率）。
- 连接池和持久性连接会阻碍负载均衡器分发连接请求的能力，举例来说，一个连接池打开了其配置数量的连接，负载均衡器把这些连接分配到了现有的 4 台 MySQL 服务器上，现在我们增加了两台 MySQL

注 9：你可以把 LVS 配置成每当应用要创建新的连接时就参与进来，LVS 毕竟不是个中间件。

注 10：在前面章节里，我们提到过一些软件的实现（Sequoia, Continuent），这里还有个语言独立的方案是 DBIx::DBCluster (<http://sqlrelay.sourceforge.net>)。

服务器。因为连接池不会再生成新的连接，那么这两台 MySQL 服务器就处于空闲状态了。而连接池里的那些连接还在那 4 台服务器中间分配着，结果就是一些服务器过载运行，而另外一些闲置着。针对这样的问题，可以通过在不同层面上设置连接池内连接的失效时间来缓解，但是，这个做法比较复杂，也难以实现。连接池解决方案只有当它们自己处理复杂平衡时才会发挥最好的效果。

- 许多多用途负载均衡器能给 HTTP 服务器做健康和负载检查。一个简单的负载均衡器就能校正服务器在 TCP 端口接受的最小连接数，一个更好一点的负载均衡器就能自动发起一个 HTTP 请求，然后通过检查其响应代码来确定这个 Web 服务器是否运行正常。MySQL 服务器在 3306 端口上不会接受任何 HTTP 请求，因此需要你自己来构建一个健康检查方法。可以在 MySQL 服务器上安装一个 HTTP 服务器软件，然后编写个脚本，让负载均衡器来检查 MySQL 服务器的状态，并返回一个适当的状态码。（注 11）其中，最需要检查的是操作系统的负载情况（一般是查看/RPOC/LOADAVG）、复制的状态和 MySQL 的连接数。

负载均衡的算法

有许多种算法来决定哪个服务器来接收下一个连接。每个厂商都用各自不同的算法，下面这个清单罗列了那些可用的方法：

随机

当请求到来时，负载均衡器从可用服务器池里随机选择一台服务器来处理。

轮询

负载均衡器按下面这个循环序列把请求转发给服务器：A、B、C、A、B、C 等。

最少连接优先

下一个连接会被转发到正处理着最少连接的服务器。

最快响应优先

把下一个连接分派给处理请求最快速的服务器。当服务器池里的服务器处理速度有快慢之分时，这个策略就相当有用。然而，当查询复杂度变化较大时，这样的算法就有点难以应付了。即使同样的查询在不同的环境运行也有不同的表现，例如当同样的查询正处于查询缓存上时，或者要查询的数据正好在服务器缓存里时。

散列化

负载均衡器把连接的源 IP 地址散列化，然后映射到服务器池里的某一台服务器上。每次有来自相同 IP 地址的连接，都会被分派到相同的服务器上。只有当池里的服务器数量发生变化时，这个绑定才会随之发生变化。

权重

负载均衡器能够同时使用以上几种算法，并通过权重来得到结果。比如说你有单 CPU 和双 CPU 的机器，双 CPU 机器的性能大概要比单 CPU 高两倍，这样你就可以告诉负载均衡器把多于两倍的请求分派给双 CPU 机器。

注 11：实际上，如果你的编码功夫能够写出一个程序去监听 80 端口，或者你把 xinetd 配置到程序中，你就用不着安装一个 Web 服务器。

哪个才是最好的负载均衡算法要取决于应用的工作量。举例来说，使用最少连接算法在有新服务器加入服务器池时，可能会使大量连接涌入到新服务器上——此时服务器的缓存还没来得及全部激活。本书第一版的作者曾经亲身体验过这样的情况。

你将需要通过实验为系统的工作量找到最好的性能状态，在日常的运行环境之外，也要考虑到一些特殊环境下的情况。即使在那些特殊的环境下——比如高负载、样式改变了、特定的几台服务器离线了，你的系统也应该能做点什么以防止发生大错误。

以上我们讨论的都是即时处理请求的算法，还没涉及将连接要求放入等候队列。有些时候那些使用了队列的算法会更有效一些。举例来说，有个算法在数据库服务器上维护着一个指定的并发性，就是说在同一时刻不会有超过 N 个的活动事务存在，如果有太多的活动事务生成，这个算法就会把一个新的请求放入队列中，然后让可用服务器列表里的第一个服务器来处理它。一些连接池就支持队列算法。

在服务器池里增加和减少服务器

在服务器池里增加一台新服务器，往往不是简单地接上一台服务器，然后告诉负载均衡器这个新服务器的到来。你可能想这样也没错，只要不被突然涌入的连接冲掉就行了。但是这不总是正确的，有时候需要慢慢地给服务器增加负载，有些服务器的缓存还是“冷”的，运转很慢，以至于一段时间它根本无法执行任何查询。

当一台服务器的缓存还是“冷”的时候，即使很简单的查询都会等待很长的时间才会结束。如果用户只是查看一个页面视图，而服务器花了 30 秒才返回所需要的数据，那么说明这台服务器还不可用，哪怕是一点点数据量。这样，你只有在通知负载均衡器加入这台新服务器之前，暂时把 `SELECT` 查询的映射到一台活跃的服务器上。可以在新服务器上读取和转发那台活动服务器的日志文件。

你应该在连接池里配置这些服务器，这样就能保留一些剩余的负载能力，在撤下服务器维护时使用，或者在其他服务器故障时派上用场。你的每台服务器都应该保持多于“足够”的负载能力，以备不时之需。

要确认配置的限度足够高，即使在服务器撤出将服务器池时还是继续工作的。举例来说，如果发现每台 MySQL 服务器典型的连接数为 100，那你就应该将池里的每台服务器的 `max_connections` 设置为 200。这样一来，即使池里的半数服务器都故障了，池还能继续处理同样数量的连接。

9.3.3 主服务器和多台从服务器之间的负载均衡

Load balancing with a master-slave MySQL Server

最常用的复制技术是一台主服务器和多台从服务器，在这个架构上，它们都不可或缺。许多应用都假设只有一个目标地址做所有的写操作，或者假设所有数据在某台服务器一直是可用的。虽说这个架构不太具有扩展性，但还是可以通过一些方法使负载均衡也能在它上面发挥良好的作用。下面的内容就是对其中的一些方法做一次浏览：

功能分区

通过为一个特别的用途配置从服务器或从服务器组，也能使负载能力获得一点点的提升。对于常见的那些功能，一般都可以把它们分为报表、分析、数据仓库和全文搜索。在本书 371 页的“自定义复制方案”你可以看到更多的内容。

过滤和数据分区

使用复制过滤技术，可以在各类似的从服务器中间把数据分区（查看 360 页的“复制过滤”）。如果主服务器上的数据已经分离到各个数据库或各个表里，这样做起来就更好了。不幸的是，没有一种内置的过滤方法在单独行的层次上实现这个目的。然而，也可以用这样的方法来实现行级别的过滤：将数据复制到一台分布式主服务器，使用带触发器的 Blackhole 表根据列值把每一行数据插入到不同的表里。

甚至可以采用更奇特的方法，比如复制到 Federated 表里，但是这可能会弄乱数据。Federated 表引入了内置服务器依赖技术来避免这个问题。

即使不把数据分区到各个从服务器上，你也可以通过读操作分离而不是随机分配来提高缓存的效率。举例来说，你可以将名字首字母为 A-M 的用户的所有读操作都分派到其中的一台服务器，而 N-Z 的分派到另外一台服务器。这样就能充分利用两台服务器的缓存，因为每次读数据时，更容易在缓存里找到相关的数据。最好的情况是，没有任何写操作，这样能使用的缓存相当于是两台服务器的缓存之和。作为对比，如果你随机地分配读操作到各个服务器上，那每台服务器的缓存里的数据都几乎一样，所以无论你有多少台从服务器，总的有效缓存数跟一台从服务器一样多。

将部分写操作移到从服务器上

主服务器未必总是处理所有的读操作。你可以从主服务器移走相当数量的冗余工作量到从服务器上，通过分解写操作，把其中一些步骤放到从服务器上去执行。更多内容请查看 299 页的“复制的过度时延”。

保证从服务器跟上更新速度

如果想在从服务器上运行某一个进程，它需要及时知道数据当前在哪个同步点上——哪怕需要等待一会儿后才会到那个点上——你可以使用 `MASTER_POS_WAIT()` 函数让这个进程等待，只到从服务器赶上主服务器上的那个同步点。另一个办法是，可以使用复制里的“心跳”技术让从服务器定时检查数据更新情况，不过它不提供秒级以下的时间粒度。更多内容请查看第 379 页的“衡量从服务器的时延”。

447 同步写入

你也可以使用 `MASTER_POS_WAIT()` 来确保写操作执行到一台或多台从服务器上。如果应用为了保证数据安全性，需要尽量赶上数据更新，那就可以在各个从服务器上轮流运行 `MASTER_POS_WAIT()`。这就像创建了一个同步围栏，需要很长时间才会轮一圈，即使各个从服务在数据同步上落后很多。这个办法可以在相对必须的时候采用（你也可以一直等待，直到一台从服务器收到事件信号，如果你的目标只是为了确保某台发出事件信号）。

9.4 高可用性

High Availability

许多人认为如果系统能响应用户的操作就是可用的。然而，可用性要比这个更复杂一些。一个应用是可以响应的，但是它可能处于“降级”模式了，它的一部分功能已经出了故障，只是内部的容错能力让它继续在运行。在做维护或其他特定情况下，你也可以让应用在只读模式下运行。举例来说，有着很多用户的图片共享网站上，用户不会介意总有一段时间里无法上传新照片；但另外一方面，ATM 机用户不希望看到屏幕上显示着“维护性只读模式”。所以，在可用性方面，网站可以这样，ATM 机就不能。

实现高可用性其实也很简单：建立冗余机制，当系统某部分故障时，能在线进行替换。其中，困难的是完成这个过程要快速且可靠。

9.4.1 高可用性规划

Principles for High Availability

各个应用都有不同的可用性需求。在你开始为某一个正常运行时间的目标而全力以赴之前，要问一下自己：到底什么才是你真正想要达到的目的？可用性每提高一次，它的花费都会超过前一次，可用性的效果和费用是成非线性比例关系的。

高可用性里的最重要原则是发现和消除系统里那个会导致全盘崩溃的最薄弱的环节。在头脑里想象你的整个应用，然后试着找出所有这样的环节：它是一块硬盘？一台服务器？一台交换机或者是一台路由器？或者是某个机架上的电源？你的所有机器都放在一个数据中心吗？或者“冗余”数据中心是同一家公司提供的吗？系统内那些没有冗余的部分都会是这样的薄弱环节。还有一些薄弱环节依赖于具体的服务，比如 DNS、单个网络服务供应商（你可以在机器上查看冗余网络连接是不是真的连到不同的骨干网上的）、单独一个电网。

试着去了解所有影响可用性的组件，做一下风险权衡，然后从那些风险最大的组件入手。一些人耗费大量力气在那种可以处理所有硬件故障的软件上，但是，这种软件上的 Bug 会导致更长的停工时间，这远胜于它所节省的时间。还有些人用各种冗余技术来建造“永不沉没”的系统，但是他们忘了数据中心那边可能会断电或者断网，或许他们也忘了恶意攻击者和程序员错误地删除或破坏了数据——比如不小心执行了 DROP TABLE——这些都会导致停工时间。

可以通过风险曝光系数来找出高优先级的风险因素，所谓的风险承担系数=故障的发生概率×故障带来的损失。画一张简单的表格，在纵列上依次是可能性、损失和曝光系数，这样就很容易找出你要优先处理的项目了。

你无法消除所有的“死穴”（Single Point of Failure, SPoF）。增加一个冗余组件并非总是可行的，因为有些限制条件无法克服，比如地理位置、预算和时间等约束。

下一步，假想一下当系统因故障、升级或应用更新时，你把系统切换（或失效转移）到了一个备用系统上。这时，任何导致应用不可用的部分都需要建立一个失效转移计划，同时，你也需要知道一次失效转移需要多长的时间。

一个相关的问题是在一次故障转移后，你替换一个故障的组件需要多长时间。除非恢复了系统的剩余能力，否则你会有更少的冗余和更多的风险。然而，有个备用系统并不是说就用不着及时地替换故障组件。让你准备一台备用服务器，装上操作系统，加载上最新的数据，一共需要多少时间？你有足够的备用服务器吗？你所需的应该不止一台。

另外一个要考虑的事情是：即使应用离线后，你会丢数据吗？如果一台服务器遭遇到灾难性故障时，你总会丢失一些数据，比如最近的那些事务，它们刚写入到二进制日志里，还没来得及写到从服务器的转发日志上。你能容忍这样的情形吗？许多应用都可以。对此的替代性方案往往是昂贵的、复杂的，还有更多性能上的开销。举例来说，你可以采用 Google 的同步复制补丁（接下去会更多地提到它），或者把二进制日志放在一台独立的设备上，它通过 DRBD 来复制，这样即使系统彻底故障了，你也不会丢失数据。

一个聪明的应用架构常常能减少你在可用性上的需求，至少对系统的一部分而言是这样的，于是，高可用性的目标就更容易达成了。在你应用中把关键和非关键部分分离开来，能帮你节省大量的工作和金钱，因为为更小的系统提供冗余和高可用性会更容易做到。

通常来讲，过了某一点后，为应用保持高可用性、防止数据丢失会很难而且花费昂贵，所以，我们建议你设立一个现实的目标，以免这方面的工作做过头。幸运的是，对于大多数应用而言，能达到 99% 或 99.9% 的正常工作时间已经算是高可用性了。

449 9.4.2 增加冗余

往你的系统里增加冗余有两种形式：增加备用的能力和组件的复制。

事实上增加备用的能力很容易做到——你可以使用任何在本章中提到过的技术。增加可用性的一条途径是创建服务器集群或者服务器池，并增加一个负载平衡方案。当一台服务器故障时，其他服务器可以承担它的负载。还有个好办法是尽可能地让组件留有性能的余地，当服务器因为负载增加，或组件故障时，就有足够的提升空间来应付此时的性能问题。

从各方面来考虑，你需要为组件准备些备用品，当该组件故障时，它能立即顶上。要做个副件很容易，像网卡、路由器或硬盘——及其他所有你想到的很容易坏掉的设备。

要复制整个 MySQL 服务器就有点难了，因为服务器没有数据就没什么用。这意味着你的备用服务器也能访问到主服务器的数据。接下来我们要讨论就是如何做到这一点。

共享存储架构

共享存储是消除一些系统“死穴”的方法之一，常常与 SAN 一起使用（存储区域网，在第 325 页有更多介绍）。在这个策略里，活动的服务器挂接上文件系统，并完成正常操作。如果这台服务器宕机了，那备用服务器就能马上接上同一个文件系统，做一些必要的恢复操作，然后基于刚才那台服务器的文件之上，启动 MySQL。这个过程在逻辑上跟修复那台故障的服务器没什么两样，除了一点就是这个过程的速度要快很多，因为备用服务器已经处于启动状态，随时可以运行。文件系统检查和 InnoDB 的恢复是你可能遇到的最大延时。

共享存储也能用于消除一些数据丢失的场景，但它仍然作为一个“死穴”存在。如果它倒下了，那整个系统也倒下了。如果它故障时损坏了数据文件，那备用服务器也没法将它们恢复过来。我们强烈推荐使用 InnoDB 或别的事务性存储引擎来做共享存储。服务器崩溃最容易损坏的是 MyISAM 表，而修复它们要花很长的时间。

复制磁盘架构

复制磁盘技术是另一种在主服务器灾难性故障时保证数据安全的方法。在 MySQL 中，使用最多的磁盘复制技术是 DRBD (<http://www.drbd.org>)，将它和来自 LINUX-HA 项目的工具一起使用（接下来会有更多关于它的内容）。

450 DRBD 是个同步的，块层次的复制技术，它以 Linux 核心模块的形式来实现，它从一个主设备通过网卡将每一块数据复制到另外一台服务器上的块设备（第二个设备），并在把块数据提交到主设备之前把它写下来（注 12）。

DRBD 只能在主动—被动模式下运行。被动设备是热备用机，你无法访问到它——不只是在只读模式下——除

注 12：实际上你能调节 DRBD 的同步层次。可以将它设为异步的，让它一直等待，直到远程设备上收到了数据，或者让它一直堵塞，直到远程设备把数据写入到磁盘里。在此，强烈建议给 DRBD 配备一块网卡。

非它成为主机后。因为在第二设备上的写必须要先于在主设备的写，所以，第二设备的性能至少要跟主设备一样好，否则，它就会限制在主设备上写的性能。另外，如果使用 DRBD 来做一个可互换的备用机，那这台备用机的硬件配置必须要跟主服务器一样。

如果活动服务器出故障了，可以将第二设备提升为主设备。因为 DRBD 复制磁盘时是在块层次上完成的，而文件系统又可能变得不稳定，这就意味着最好是使用一个日志文件系统来做快速恢复。一旦服务器恢复了，接下来可能就是 MySQL 做自身的恢复了。如果第一服务器在恢复，它会假定自己是第二服务器的身份，与目前的主设备进行同步。

从实际的故障恢复过程而言，DRBD 跟 SAN 很相似：有一个热备用机器，它从故障机器那里取得相同的数据后开始工作。两者之间最大的差别是 DRBD 是复制存储——不是共享存储——所以，在 DRBD 里，你使用的是数据的一个复制本，而 SAN 使用的是同一个设备上的同一份数据。在以上两种情况下，当启动备用机上的 MySQL 时，它的缓存都是空的。相比之下，一个复制从服务器里有部分缓存可能已经“预热”了。

DRBD 有一些不错的结构和能力能防止集群软件常会遇到的问题。一个例子是“裂脑综合症”，这发生于两个节点同时提升为主服务器时。你可以通过配置 DRBD 来避免这种裂脑综合症的发生。然而，DRBD 也不是一个能满足每一个需求的完美解决方案。下面就让我们看一下它的缺点：

- DRBD 的故障恢复不是次秒级的。它至少需要 5 秒钟把第二设备提升为主设备，这还不包括必需的文件系统恢复和 MySQL 恢复。
- 因为必须在主动—被动模式下运行 DRBD，这使它显得很昂贵。当热备用服务器复制设备处于被动模式时，它不能被用来完成其他任务。这算不算是个缺点要取决于你的出发了。如果你真正需要高可用性，无法容忍主机故障时服务处于降级状态，但你也不能在其中一台服务器上运行两台服务器的负载量，因为如果这么做了，当它们中的一台故障时，你就无法处理这些负载了。你可以用备用机做些别的用途，比如复制从服务器，但是，你仍然是浪费了一些资源。
- 在这里，MyISAM 表特别地无用，因为它们需要很长的时间来检查和修复。MyISAM 在任何需要高可用性的系统里都不是个好选择，代之以 InnoDB 或别的存储引擎，它们会有很好的恢复性能。
- 它无法替代备份。如果是因为恶意干扰、过失、Bug 或硬件故障等原因你的数据被损坏了，DRBD 也帮不上忙：复制的数据只是被损坏的源数据的一个完美拷贝而已。你需要自己做备份（或者是 MySQL 延时复制），以此保证数据远离刚才提到的那些问题。

451

我们喜欢把 DRBD 用来对存储有二进制日志的服务器做复制。如果活动节点处故障了，你就可以在被动节点上启动日志服务器，然后使用这些恢复了的二进制日志，把从服务器设定到最近的二进制日志位置上（更多内容请查看第 374 页的“创建日志服务器”）。接下来，你就可以选择一个从服务器，把它升级为主服务器来替换崩溃的系统。

MySQL 同步复制

在同步复制中，主机上的事务直到把数据提交到一台或多台从服务器之后才会结束。同步复制也分好几个层次，它们都有各自常用的名字。MySQL 在本书写作的时候，还没提供同步复制功能，但是有许多第三方解决方案可供选择。其中一个就是 Google 的内部补丁。

Google 拥有大量的 MySQL 和 InnoDB 的补丁，给它们增加了许多额外的功能。其中一个就是半同步复制，主机在启动事务后，直到有一台从服务器接收到这个事件才会结束。Google 已经发布了 MySQL 4.0.26 和 5.0.37

的补丁。你可以在 <http://code.google.com/p/google-mysql-tools> 下载到这些补丁和相关工具。

还有个可选方法是 Solid Information Technology 公司的高可用性技术，它将此功能移植到了 solidDB for MySQL 里。这个解决方案在 MySQL 复制时，有几个优点：

- 从服务器不会落后于主服务器。
- Solid 在从服务器上使用多线程来写数据，在许多场合下提高了复制的性能。
- 用户可以自己配置在“安全”层次上的复制。在 1-SAFE 模式下，事务在主机上提交后就立即返回；2-SAFE 模式下，事务直到在从服务器上提交成功后才返回，并提供了多一层的安全保障来应付崩溃发生时的情况。

然而，这个方案只能用在 solidDB 存储引擎下，无法与 MyISAM \ InnoDB 及其他存储引擎一起使用。Solid 公司在将来可能会把更多的高可用性技术移植到 MySQL 上来。

除了以上两种 MySQL 自带的方案外，也可以使用中间件解决方案，比如 Continuent。

9.4.3 故障转移和故障恢复

Failover and Failback

故障转移是移除故障服务器，并用另外一台服务器代替的过程。在高可用性架构里，这是最重要的一部分。

在内容展开之前，让我们来定义一个新术语。在规范的情况下，我们是使用“故障转移”，而有些人把“故障切换”当作了它的同义词。有时，人们也说“主备切换”，它指的是按计划进行的主机和备用机的切换，不是故障之后的应对措施。

我们也使用术语“故障恢复”来说明故障转移后的服务器回迁过程。如果系统具备了故障切换能力，那么故障转移就是一个双向过程：在服务器 A 崩溃后，服务器 B 就替代了它；然后你修复了服务器 A，再把服务器 B 替换回来。

故障转移的缘由各不相同。我们已经在上文中讨论过它们中的一些了，因为负载均衡跟故障转移在许多方面是相似的，它们之间的分界线有点模糊。一般来说，我们想到的一个完整的故障转移方案，它在最小程度上至少能够监控服务器的运行，并在其崩溃时自动地替换掉它。这个过程对于应用而言，应该是透明。负载均衡不需要提供这样的功能。

在 Unix 的世界里，故障转移经常是使用 High Availability Linux 项目 (<http://linux-ha.org>) 提供的工具来完成的。这个项目——暂且不管它创建者的名字——是运行在许多类 Unix 操作系统上的。“心跳”工具提供了监控的功能，其他不同的工具完成 IP 接管和负载均衡等功能。你可以将它们与 DRBD 和（或）LVS 组合起来使用。

故障转移的最重要部分是故障恢复。如果服务器不能切换自如，那么故障转移就是个死胡同，只能加长停工时间。这就是我们喜欢对称复制拓扑学的原因，比如双主机配置，我们不喜欢用三台或更多的伴随主机来做环状复制。如果配置是对称的，那么故障转移和故障恢复在相对的方向上是一样的操作（在此值得一提的是 DRBD 内建了故障恢复的功能）。

在一些应用里，故障转移和故障恢复需要尽可能地快速和原子性。即使当无法达到这个要求时，你仍然最好不要让事情依赖于那些不可控制的因素上，比如 DNS 更改或者应用配置文件。一些最糟糕的问题只有当系统变得很庞大的时候才会显现出来，比如因为某些因素需要重新启动应用时，原子性的需要才开始让你感到困惑。曾

经在许多服务器上做过原子性的代码升级的人都知道这个很困难。

因为负载平衡和故障转移两者联系很近，有些硬件或者软件都是同时为这两个目的而设立的，我们建议你选择负载平衡技术时最好让它同时带有故障转移的能力。这也是我们说要避免用 DNS 和修改代码来做负载平衡的真实原因。如果你采用了这样的策略，就有额外的工作要做了；当你提高了高可用性之后，不得不重写那部分受到影响的代码。

下面部分我们要讨论一些常用的故障转移技术。

提升一个从服务器或者切换角色

将一个从服务器提升为主服务器，或者在主—主复制设置里切换活动和非活动角色，这些都是 MySQL 故障转移方案里很重要的部分。具体操作的细节请查看第 382 页的“改变主服务器”。

虚拟 IP 地址或 IP 接管

你可以通过把一个逻辑 IP 地址指定给一个要提供特定服务的 MySQL 实例来达到高可用性的目的。如果这个 MySQL 实例崩溃的了，你就把这个 IP 地址指向另外一个不同的 MySQL 服务器。这个核心思想跟先前我们写到过的内容相同（请查看第 441 页的“移动 IP 地址”），唯一不同的是现在用它来做故障转移，而不是负载平衡。

这种方法的好处是它对于应用而言是透明的。虽然它会退出当前连接，但是，不需要你去改变应用的配置。移动 IP 地址的过程也可以自动完成，因此，所有应用都能同时看到这一改变。当服务器在可用和不可用状态之间震荡时，这一特性尤其重要。

它的不足之处如下所述：

- 你需要在同一网段里定义好所有 IP 地址，或者使用网络桥接。
- 更改 IP 需要的系统的 root 权限。
- 有时需要更新 ARP 缓存。有些设备会将 ARP 信息存储很长时间，无法在你切换 IP 后立即把 IP 地址绑定到新的 MAC 地址上。
- 你要确保网络硬件设备支持快速的 IP 接管。有些硬件需要 MAC 克隆后才能正常工作。
- 有些服务器在功能部分损坏的情况还会继续占用 IP 地址，这样就需要你从物理上将其关闭或者从网络上断开。

浮动 IP 地址和 IP 接管可以很好地应付出现在两台临近（即同一子网里）的机器之间的故障转移。

等待变化传播开去

经常有这样的情况：当你在某一层定义了冗余之后，你不得不等待更低级的那层对此作出相应的改变。在本章前面的篇幅里，我们指出通过 DNS 来改变服务是种虚弱的方案，因为 DNS 的改变传播很慢。虽然改变 IP 地址给你更多的控制，但是在局域网里 IP 的改变还是要依靠更低级的层——ARP 来传播这一变化。

MySQL 主—主复制管理器

MySQL 主—主复制管理工具 (<http://code.google.com/p/mysql-master-master>), 或者简称为 mmm, 是一系列脚本, 用来执行监控、故障转移和主—主复制配置管理等任务。尽管它的名字如此, 但是它也对其他拓扑架构的服务器做自动的故障转移处理, 比如简单的主—从、主—主下的一个或多个从服务器。它使用抽象的角色概念, 比如读者或作者, 混合了固定 IP 和移动 IP。当它发现一台服务器故障时, 它会按照需要把 IP 地址指定到另外一台服务器, 从而变换了角色。它也能用在规划维护性故障转移或其他任务上。

这个工具通常的安装方法是建立一对副的主 MySQL 服务器, 每台上面运行着 mmm_agent 进程。你需要为它们逐个配置好 IP 地址、用户名、密码等信息。每个 mmm_agent 进程只知道自己所在的点。

这个方案里还有个单独的监控节点存在, 它的硬件配置用不着跟任何一个一样。它监视了那两个节点, 并处理故障转移——就是说移动“作者”角色。这样, 总共会有 3 个虚拟 IP 地址连接到 MySQL 服务器: 两个是“读者”角色, 一个是“作者”角色。可以用 mmm_control 程序显示和控制 MySQL 实例, 并根据实际需要来移动“作者”角色。

你可以将 mmm 跟其他技术混着来用 (比如前面我们提到过的 Google 的半同步复制补丁), 来进一步提高可用性和可靠性。

中间件解决方案

可以使用代理、端口转发、NAT、硬件负载均衡器等来处理故障转移和故障切换。然而, 它们也把自己作为“死穴”引入到了系统里, 你需要为它们准备冗余设备来避免这样的问题。

这个方案里比较好的一点是对应用来说, 远程数据中心就像在同一网络里似的。这使得你可以用浮动 IP 这样的技术让应用与完全不同的数据中心发起通信。你可以配置每个数据中心里的每一个应用服务器, 让它们通过自己的中间件进行连接, 这样, 它们就会访问连接都路由到活动的数据中心。图 9-7 描述了这个配置。

455

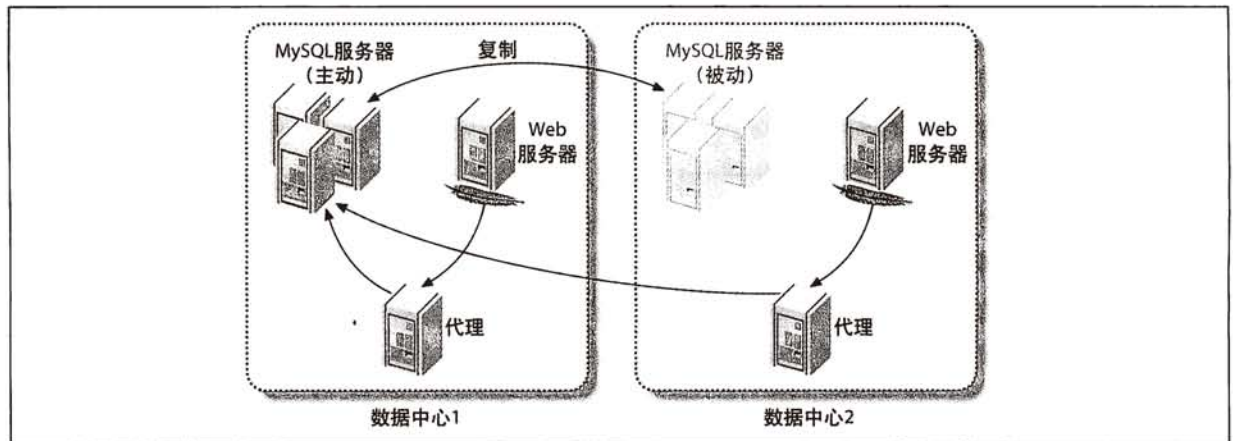


图 9-7: 使用 MySQL Proxy 在各数据中心间分派连接

如果这个活动数据中心的 MySQL 彻底崩溃了, 中间件就会把访问连接定向到另一个数据中心的服务器池, 而应用用不着随这个变化而改变。

这个配置方法的主要缺点是在一个数据中心的 Apache 服务器与另一个数据中心里的 MySQL 服务器之间的时延比较大。为了缓解这个问题，你可以把 Web 服务器设置为重定向模式。这样，连接都会被重定向到那个放置了活动 MySQL 服务器池的数据中心那里。你也可以使用 HTTP 代理来完成这个事情。

图 9-7 里显示了 MySQL 代理连接到 MySQL 服务器的情形，但是你可以将这个方案与其他中间件架构结合使用，比如 LVS 和硬件负载均衡器。

在应用中处理故障转移

有时候，让应用来处理故障转移会更简单更有弹性。举例来说，如果应用遇到一个错误，而外部观察者在正常情况下无法检测到这个错误，比如表明数据库数据损坏的一个错误消息，那么应用就能自己来执行故障转移过程了。

虽然把故障转移处理过程集成到应用中看上去很吸引人，但它往往运作得不好。许多应用都有着大量的组件，比如 cron 任务、配置文件、用不同语言写就的脚本。

因此把故障转移集成到应用里的做法用得并不广泛，特别是当应用逐渐增长，变得更加复杂的时候。

然而，把监控功能构建在应用里倒是个很好的主意，当它发现有需要，它就能**立即开始**故障转移过程。应用应该也能够管理用户体验，比如提供将功能降级的操作功能，显示相应的消息给用户。

应用层面的优化

本书若不讲解一章关于连接到 MySQL 的应用程序优化的内容，那就不能算完整，因为人们常常把一些性能方面的问题都归咎到 MySQL 身上。书里面我们更多地是讲到 MySQL 的优化，但是，我们不想让你错过这个更大的图景。一个糟糕的应用设计会使你无论怎么优化 MySQL 也弥补不了它带来的损失。实际上，有时候对于这类问题的答案是把它们从 MySQL 上脱离开来，让应用自己或其他工具来做这些事情，这样或许会有较好的性能表现。

本章不是构建高性能应用的参考书，我们只是希望通过阅读这一章让你避免那些常见的会伤及 MySQL 性能的小错误。下文中我们以 Web 应用为主要讲解对象，因为 MySQL 主要是用在 Web 应用上的。

10.1 应用程序性能概述

对于更快性能的追求开始时很简单：应用响应请求花费了太长的时间，你总要为此做点什么吧。然而，真正的问题是什么呢？通常的瓶颈是缓慢的查询、锁、CPU 饱和、网络延时和文件 I/O。如果应用配置错误，或者不恰当地使用资源，以上任何一个因素都会引出一个大问题。

10.1.1 找出问题的根源

第一个任务是找出“肇事者”。如果你的应用具备了显示系统运行概况的功能，这做起来就简单了。如果你已经做到了这一步，但还是没法找出引起性能低下的原因，那你就需要增加更多的概况信息的调用，去找出那些要么缓慢要么被多次调用的资源。

如果你的应用因为 CPU 高占用率而一直等待，并且应用里有高并发性，那我们在第 55 页的“分析应用程序”所提到过的“丢失的时间”可能就成了问题了。鉴于此，有些时候在有限的并发条件下生成应用的概况信息是很有用的。

网络延时会占用大块的时间，哪怕是在局域网里。应用层面的概况信息已经包括了网络延时，因此，你应该在概况系统里看到网络往返延时带来的影响了。举例来说，如果一个页面执行了 1 000 个查询，即使每次只有 1 毫秒的延时，那累加起来也有 0.5 秒的响应时间，这对高性能应用来说已经是个很大的数目了。

如果应用层面概况信息收集得很充分，那就不难找出问题的根源。如果还没有内置概况功能，那就尽可能地加上它。如果你无法添加这个功能，那也可以试试第 76 页的“当你无法加入概况信息代码时”里提供的那些建议。这个总比钻研像“什么引起应用变慢”那样没头绪的理论设想要更快更容易。

10.1.2 寻找常见问题

Look for Common Problems

同样的问题我在应用里一次又一次地遇到，其原因往往是人们使用了设计糟糕的原有系统，或者采用了简化开发的通用框架。虽然这在某些时候能让你在开发一些功能时变得方便又快速，但它们也给应用增加了风险，因为你不知道它们底下是怎么工作的。这里有一张清单你应该逐个检查一下：

- 在各个机器上的 CPU、磁盘、网络 and 内存资源的使用情况如何？使用率对你而言是否合理？如果不合理，就检查那些影响资源使用的应用的程序基础。配置文件有时就是解决问题的最简单方法，举例来说，如果 Apache 耗光了内存，那是因为它创建 1 000 个工作者进程，每个工作进程需要 50MB 内存，这样，你就可通过配置文件配置这个应用能申请的 Apache 工作者进程数。你也可以配置系统，使之创建进程时少用些内存。
- 应用是否真正使用了它所取得的数据？一个常见的错误是：读取了 1 000 行数据，却只要显示 10 行就够了，其他 990 行就丢弃了（然而，如果应用缓存了余下的 990 条记录供以后使用，那么这可能是特意做的优化）。
- 应用里是否做了本该由数据库来做的处理？反之亦然。有个对应的例子是：读取了所有行的数据，然后在应用里计算它们的总数；以及在数据库里做复杂的字符串处理。数据库擅长于计数，而应用的编程语言擅长于正则表达式。你该使用正确的工具去干正确的活。
- 应用里执行了太多的查询？那些号称能“把程序员从 SQL 代码里解救出来”的 ORM (Object-Relational Mapping) 就因此常被人们责备。数据库服务器是被设计用来匹配多表数据的，因为要移除那些嵌套循环，代之以联接 (Join) 来做同样的查询。
- 应用里执行的查询太少了？我们只知道执行了太多的查询会成为问题。但是，有时“手工的联接”和与其相似的查询是个好主意，因为它们可以更加有效地利用缓存，更少的锁（尤其是 MyISAM），有时当你在应用的代码里使用一个散列联接时（MySQL 的嵌套循环的联接方法往往是低效的），查询的执行速度会更快。
- 应用是不是在毫无必要的时候还连到 MySQL 上去了？如果你能从缓存里读取数据，就不要去连数据库了。
- 应用连接到同一个 MySQL 实例的次数是不是太多了？这可能是因为应用的各个部分都各自开启了自己的数据库连接。有个建议在通常情况下都很对：从头到尾都重用同一个数据库连接。
- 应用是不是做了太多的“垃圾”查询？一个常见的例子是在做查询前才去选择需要的数据库。一个较好的做法是连接到名称明确的数据库，并使用表的全名做查询。（这样做，也便于通过日志或 SHOW PROCESSLIST 去查询情况，因为你可以直接执行这些查询语句，无需再更改数据库）。“准备”数据库连接又是另一个常见的问题，特别是 Java 写的数据库驱动程序，它在准备连接时会做大量的工作，它们中的大多数你都可以关闭。另一种垃圾查询是 SET NAMES UTF8，这纯粹是多此一举（它无法改变客户端连接库的字符集，它只对服务器有影响）。如果你的应用已确定在多数任务下使用的是某一个字符集，那你就可以避免这样无谓的字符集设置命令。
- 应用使用连接池了吗？这既是好事情也是坏事情。它限制了连接的数量，这在连接上查询数不多的情况下 (Ajax 应用就是个典型的例子) 是有利的；然而，它的不好的一面是，应用会受限于是使用事务、临时表、连接指定的设置和定义用户变量。

459

- 应用使用了持久性连接吗？这样做的直接结果是会产生太多的数据库连接到 MySQL 上。通常情况下，这是个坏主意，除了一种情形：由于慢速的网络导致 MySQL 的连接成本很高，如果每条连接上只执行一两个快速的查询，或者频繁地连接到 MySQL，那样你会很快用完客户端的所有本地端口（更多内容请查看第 328 页的“网络配置”）。如果你正确地配置了 MySQL，根本不需要持久性连接，可以使用“跳过名称解析”来防止 DNS 的查找，并确认该线程的优先级足够高。
- 即使没有使用，应用是不是还打开着连接？如果是，特别是当这些连接连向多台服务器时，它们可能占用了其他进程需要的连接。举例来说，假设你连接到 10 台 MySQL 服务器。由一个 Apache 进程占用 10 个连接数，这不是个问题，但是它们中只有一条连接是在任何指定时间里做着一些操作，而其他 9 条连接绝大多数时间都处于睡眠状态。如果有一台服务器响应变得迟缓，或者网络延时变长，那其他几台服务器就遭殃了，因为它们根本没连接可用。对于这个问题的解决办法是控制应用使用数据库连接的方式。

举例来说，你可以在各个 MySQL 实例中依次做批量操作，在向下一个 MySQL 发起查询前，关闭当前的所有连接。如果你要的是时间消耗很大的操作，比如调用一个 Web Service，可以先关闭与 MySQL 的连接，等这个耗时的调用完成后，再打开 MySQL 的连接，完成剩余的需要在数据库上操作的任务。

持久性连接与连接池的不同点比较模糊。持久性连接有与连接池相同的副作用，因为在各种情况下重新使用的连接往往都带有状态。

然而，连接池并不总是导致许多连接到服务器的联接，因为它们是队列化的，并在各进程间共享这些连接。在另一方面，持久化连接是基于每个进程来创建的，无法被其他进程所使用。

与持久性连接相比，连接池在连接策略上有更多的控制。你可以把一个连接池配置成自动扩充的，但是通常的做法还是当连接池满的时候，新的连接请求都被放在队列里等待。这使得这些请求都在应用服务器上等待，总好过 MySQL 因为连接太多而超载。

有太多的方法使查询和连接更加快速，一般性准则是避免把它们放在一起，胜于试着把它们加速。

10.2 Web 服务器的议题

Web Server Issues

Apache 是 Web 应用中使用最广泛的服务器软件。在各种用途下，它都能运行良好，但如果使用得不恰当，它也会占用大量的资源。最常见的一个情况是让它的进程活动了太长的时间，并把它用在各种不同类型的任务下却没有做相应的优化。

Apache 经常在 prefork 配置项里使用 mod_php、mod_perl、mod_python。预分叉（Prefork）是为每个请求分配一个进程。因为 PHP、Perl 和 Python 等脚本语言运行起来很费资源，每个进程占用 50MB 或 100MB 内存的情形也不罕见。当一个请求处理完后，它会把绝大多数内存归还给操作系统，但不会是全部。Apache 会让这个进程保持在运行状态，以处理将要到来的请求。这就意味着如果这个新来的请求只是为了获得一个静态文件，比如一个 CSS 文件或一张图片，你都需要重新启用那个又“肥”又“大”的进程来处理这个简单请求。这也是为什么把 Apache 用作多用途 Web 服务器是件危险的事情。它是多用途的，若你对它进行了有针对性的配置，它才会有更好的性能表现。

另外有个主要的问题是如果你打开了 Keep-Alive 参数项，进程就会长时间地保持忙碌状态。即使你不这么做，

有些进程也会这样。如果内容是像“填鸭”一样传给客户端的，那这个读取数据的过程也会很漫长（注1）。

人们也经常犯这样的错误：按 Apache 默认开启的模块来运行。你可以按照 Apache 使用手册里的说明，把你不需要的模块都关闭掉，做法也很简单：查看 Apache 的配置文件，把不需要的模块都注释掉，然后重启 Apache。你可以从 php.ini 文件中把不需要的 PHP 模块都移除。

如果你创建了一个多用途 Apache 才需要的配置当作 Web 服务器来用，你最后可能会被众多繁重的 Apache 进程所拖垮，这些进程纯粹浪费你的 Web 服务器上的资源。而且，它们会占用大量与 MySQL 的连接，以至于也浪费了 MySQL 的资源。这里有一些方法能给你的服务器“减负”（注2）：

- 不要把 Apache 用作静态内容的服务，如果一定要用，那也至少要换个另外的 Apache 实例来处理这些事情。常见的替代品有 lighttpd 和 nginx。
- 使用一个缓存代理服务器，比如 Squid 或 Varnish，使用户请求无须抵达 Web 服务器后才能被响应。即使在这个层面上你无法缓存所有的页面，你也能缓存大部分页面，并通过 Edge Side Includes (ESL, <http://www.esi.org>) 技术把页面上的小块动态部分放到缓存的静态部分里。
- 对动态内容和静态内容都设置过期策略。你可以使用缓存代理软件，像 Squid，去验证内容的明确性。Wikipedia 就是用这样的技术在缓存里移除内容已发生变化的文档。
- 有时你可能需要改变一下应用，使它能使用更长的超期时间。举例来说，如果你告诉浏览器要永久缓存 CSS 和 JavaScript 文件，然后又对这个网站静态 HTML 文件做了一些修改，这样这些页面的显示效果可能会变得很糟。对此，你需要使用一个唯一的文件名对每次修订后的页面文件都作一个明确的版本标记。举例来说，你可以自定义你的网站发布脚本，把 CSS 文件复制到/css/123_frontpage.css 目录下，这里的 123 就是 Subversion 里的修订号。你也可以用同样的方法来处理图片文件——不要重用原来的文件名，否则，即使你更新了文件内容，页面不会再被更新，不管浏览器要将原来的页面缓存多久。
- 不要让 Apache 与客户端做“填鸭”式通信。这不仅仅是慢，而且很容易招致拒绝性服务攻击。典型地，硬件化的负载均衡器会处理好缓存，Apache 就能很快地结束响应，然后让负载均衡器从缓存里读出数据去“喂”客户端。你也可以使用 lighttpd、Squid，或者设为事件驱动模式下的 Apache 作为应用的前端。
- 开启 gzip 压缩。现在的 CPU 很廉价，它可以用来节省大量的网络流量。如果你想节省 CPU 周期，那可以使用轻量级的 Web 服务器，比如 lighttpd，来缓存和提供压缩过的页面。
- 不要将 Apache 上的长距离连接配置为“保活”（Keep-Alive）模式，因为它会使 Apache 上臃肿的进程长时间处于运行状态。代替的方案是，用一个服务端的代理来处理“保活”的连接，使服务器免受这类客户端的伤害。如果将 Apache 与代理之间的连接方式设为“保活”，那是不错的主意，因为代理仅使用几个连接从服务器上读取数据。图 10-1 说明了以上两者的差异。

注1：这种“填鸭”式过程发生在当一个客户端发起一个 HTTP 请求，但无法立即得到请求结果时。直到得到全部数据之前，这个 HTTP 连接及对应的 Apache 进程都将保持忙碌状态。

注2：有一本关于如何优化 Web 应用的好书，名叫《High Performance Web Sites》，作者是 Steve Sounders (O'Reilly)。虽然它里面的大多数内容是从客户端的角度来讲怎样使网站运行得更快，但是他倡导的实践案例也适用于你的服务器。

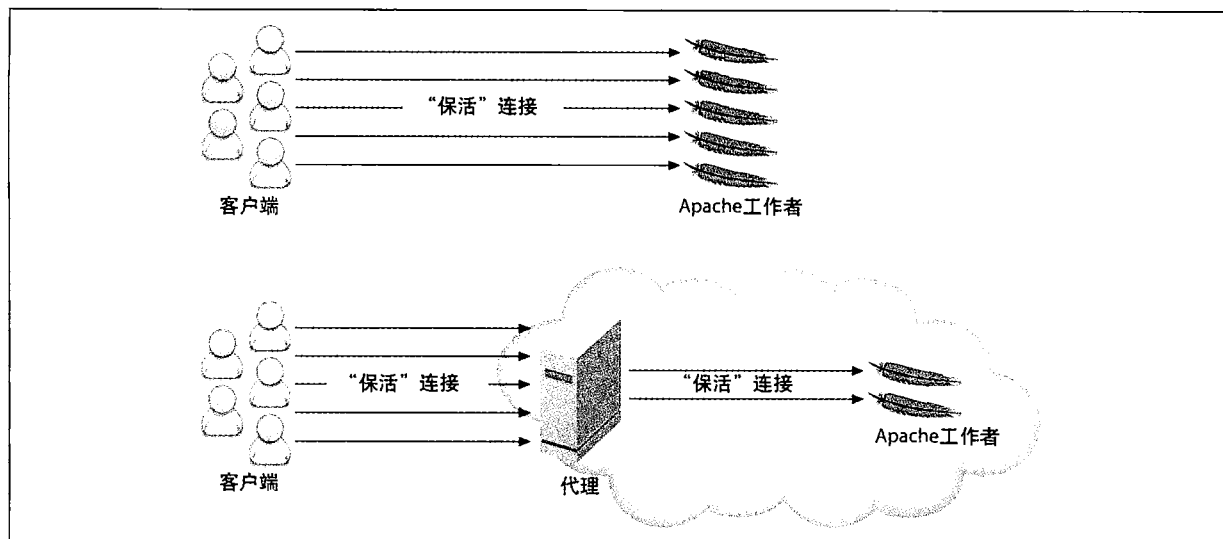


图 10-1：一个代理可以让 Apache 免受长久保持的“保活”连接的负担，从而可以使用更少的 Apache 工作者进程

以上这些策略应该可以帮助 Apache 减少进程的使用数，使你的服务器不会因为太多的进程而崩溃。然而，有些具体的操作仍然会引起 Apache 的进程长时间地运行，吞掉大量的系统资源。有一个例子就是查询外部资源时具有很高的延迟，比如访问一个远程 Web 服务器。这样的问题还是无法用上述那些方法来解决。

10.2.1 找到最佳的并发数

Finding the Optimal Concurrency

每个 Web 服务器都有它的一个**最佳并发数**——它的含义是服务器能同时处理的并发连接数目，它们既能尽可能快地处理客户端请求，又不会使服务器过载。这个“神奇的数目”需要做多次的尝试—失败的反复才能得到，相比于它能带来的好处，这还是值得一做。

对于大流量的网站而言，Web 服务器同时处理几千个连接是件很平常的事情。然而，这些连接中只有很少的一部分需要主动地去处理请求，而其他那些都是读取请求、文件上传、“喂”内容，或者仅仅等待客户端的下一步请求。

463 当并发数增加时，服务器会在某一点上达到它的吞吐量顶峰，在此之后，吞吐量会变得平稳，往往还会开始下降。更重要的是，系统的响应时间（延迟）开始增加。

想要知道究竟，就要设想如果你只有一颗 CPU，而服务器同时接收到 100 个请求，接下来会发生什么？假如一个 CPU 秒只能处理一个请求，而且你使用了一个完美的操作系统，没有任务调度的开销，也没有上下文切换的开销，那么这些请求总共需要 100 个 CPU 秒才能完成。

那么，怎样去做才是处理这些请求的最好办法？你可以把它们一个接一个放进队列里，或者对它们进行并行处理，每个请求在每一个轮回中都获得一样多的处理时间。这两种方式里，吞吐量都是每一秒一个请求。然而，如果使用队列，平均延迟有 50 秒（并发数=1），如果并行处理，那延迟有 100 秒（并发数=100）。在实际环境下，并发处理方法的平均延迟还会更高，因为其中还有个切换开销。

对于高 CPU 占有率的工作负载而言，其最佳并发数就是 CPU（或者是 CPU 里的核）的数目。然而，进程不总是可以运行的，因为它们会执行阻塞式调用，比如 I/O、数据库查询和网络请求等。因此，最佳并发数往往会多于 CPU 数目。

你可以估计最佳并发数，但是这需要精确的分析模型。通常情况下，还是通过实验的方法比较容易，你尝试着不同的并发数，然后观察系统在降低响应时间前，能达到多大的顶峰吞吐量。

10.3 缓存

Caching

缓存对于高负载的应用而言极其重要。一个典型 Web 应用里，直接提供服务要比使用缓存（包括缓存校验、作废）多生成很多内容，所以，缓存能够将应用的性能提高好几个数量级。这个技巧的关键在于找出缓存粒度和作废策略的最佳结合点。同时，你需要决定缓存哪些内容，在哪里缓存。

一个典型的高负载应用有许多层的缓存。缓存不仅仅发生在你的服务器上：它出现在整个流程的每一个步骤上，包括用户的 Web 浏览器里（这就是网页头部的有关作废设置内容的用途）。通常而言，缓存越靠近客户端，就越能节省更多的资源，更加高效。一副图片从浏览器缓存里读出要好于从 Web 服务器的内存里读取，而后者又好于从服务器的磁盘上读取。每一种缓存都有其独有的特性，比如尺寸、延时等，在接下来的章节里我们将对它们逐一进行叙述。

你可以把缓存想象成两大类：**被动缓存**和**主动缓存**。被动缓存除了保存和返回数据不做其他事情。当你从被动缓存那里请求一些内容时，它要么给你需要的结果，要么告诉你“你要的数据不存在”。一个被动缓存的例子就是 memcached。

相反地，主动缓存在找不到请求的数据时，它会做点别的事情。一般就是把你的请求传递给应用的某一部分——它能生成请求所需要的内容，然后主动缓存就会存储这部分内容，并返回给客户端。Squid 缓存代理服务器就是一个主动缓存。

当设计应用时，你总希望你的缓存是主动型（也叫**透明型**）的，因为对于应用，它们可以隐藏“检查—生成—存储”这个逻辑。你可以在被动缓存之上构建你的主动缓存。

缓存并不总是有用

你需要确定缓存是不是真地提高了系统的性能，因为它可能一点用处也没有。举例来说，在实际应用中，从 httpd 的内存中读取内容要比从缓存代理那里读取快一些。如果那个代理的缓存是建于磁盘上的，那结论会更明显。

这个原因很简单：缓存也有自己的运行开销，它们主要检查缓存的开销和提供被命中缓存内容的开销，另外还有将缓存内容作废和保存数据的开销。只有当这些开销的总和小于服务器生成和提供数据所要的开销时，缓存才有用。

如果你知道所有这些操作的总开销，你就能计算缓存能起多大的作用。没有缓存时的开销就是服务器为每个请求生成数据所需要的总开销。有缓存时的开销就是检查缓存的开销，加上缓存没命中的可能性乘以生成这些数据的开销，再加上缓存命中的可能性乘以从缓存里取出这些数据的开销。

如果有缓存时的开销小于没缓存的时候的开销，那使用缓存就可以提高系统性能，但是也不能保证肯定是这样。记在脑子里的一个例子就是从 lighttpd 内存里读取内容的开销要比代理从磁盘缓存上读取的开销要小，一些缓存总会比另外一些便宜。

10.3.1 在应用之下的缓存

MySQL 服务器有它自己的内部缓存，你也可以构建你自己的缓存和汇总表。你可以自定义缓存表，以便于更好地将它用于过滤、排序、与其他表做联接、计数，以及其他用途。缓存表比其他应用层的缓存更加持久，因为它们在服务器重启后还会继续存在。

我们在第 3 章、第 4 章里讲到过这些缓存策略，因此在本章里，我们的篇幅主要集中在应用层面和应用之上的缓存。

10.3.2 应用层面的缓存

典型的应用层面的缓存一般都是将数据放在本机内存里，或者放在网络上的另外一台机器的内存里。

应用层面的缓存一般要比更低层面的缓存有更高的效率，因为应用可以把部分计算结果存放在缓存里。因而，缓存对两类工作很有帮助：读取数据和在这些读取数据之上做计算。一个很好的例子是 HTML 文本的各个分块。应用能够产生 HTML 段落，比如头条新闻，然后将它们缓存起来。随后打开的页面里就能将这些被缓存起来的头条新闻直接放到页面上。通常来讲，缓存之前处理的数据越多，使用缓存之后能节省的工作量也越多。

这里有个不足之处就是缓存的命中率越多，要提高它而花费的钱就越多。假如你需要 50 个不同版本的头条新闻，能根据用户所在的不同地域来显示不同的头条。你需要有足够的内存来保存这全部 50 个版本的头条新闻，任何一个给定版本的头条被请求得越少，那它的作废操作也会越复杂。

应用缓存有许多种类型，以下是其中的一部分：

本地缓存

这种缓存一般都比较小，只存在于请求处理时的进程内存空间里。它们可用于避免对同一资源的多次请求。因此，它也没什么精彩之处：它往往只是应用程序代码里的一个变量或一个散列表。举例来说，如果需要显示用户名，而你只知道用户 ID，于是就设计一个函数叫 `get_name_from_id`，把缓存功能放在这个函数里，具体代码如下：

```
<?php
function get_name_from_id($user_id) {
    static $name; // static makes the variable persist
    if ( ! $name ) {
        // Fetch name from database
    }
    return $name;
}
?>
```

如果你使用的是 Perl，那么 Memoize 模块就是缓存函数调用结果的标准办法：

```
use Memoize qw(memoize);
memoize 'get_name_from_id';
sub get_name_from_id {
    my ( $user_id ) = @_;
    my $name = # get name from database
    return $name;
}
```

这类技术都比较简单，但是它们能帮你节省大量工作。

本地共享内存式缓存

这种缓存大小中等（几个 GB）、访问快速，同时，难于在各机器间同步。它们适用于小型的、半静态的数据存储。举例来说，像每个州的城市列表、共享数据存储里的分块函数（使用映射表），或者应用了存活时间（Time-to-live, TTL）策略的数据。共享内存的最大好处是访问时非常快速——一般要比任何一种远程缓存要快很多。

分布式内存缓存

分布式内存缓存的最著名的例子是 memcached。分布式缓存比本地共享缓存要大，增长也容易。每一份缓存的数据只被创建一次，因为不会浪费你的内存，当同一份数据在各处缓存时也不会引起数据一致性问题。分布式内存擅长于对共享对象的排序，比如用户信息文件、评论和 HTML 片段。

这种缓存比本地共享缓存有更高的延迟，因此最有效的使用它们的方法是“多取”操作（比如在一次往返时，读取多个对象数据）。它们也要事先规划好怎么加入更多的节点，以及当一个节点崩溃时该怎么做。在这两种情形下，应用都要决定如何在各节点间分布或重新分布缓存对象。

当你在缓存集群里增加或减少一台服务器时，一致性的缓存对于性能问题就显得尤为重要。这里有一个用于 memcached 的一致性缓存库：<http://www.audioscrobbler.net/development/ketama/>。

磁盘缓存

磁盘是慢速的，所以，持久性对象最适合做磁盘缓存。对象往往不适合放在内存里，静态内容也是（比如预生成的自定义图片）。

非常有效地使用磁盘缓存和 Web 服务器的技巧是用 404 错误处理过程来捕捉没命中的缓存。加入你的 Web 应用要在页面的头部显示一个用户自定义的图片，暂且将这个图片命名为 /images/welcomeback/john.jpg。如果这个图片不存在，它就会产生一个 404 错误，同时触发错误处理过程。接着，错误处理过程就生成这个图片，并存放在磁盘上，然后再启动一个重定向，或者仅仅把这个图片“回填”到浏览器里，那么，以后的访问都可以直接从文件里返回这个图片了。

你可以将这项技巧用于许多类型的内容，举例来说，你用不着再缓存那块用来显示最新头条新闻的 HTML 代码了，而把它们放入一个 JavaScript 文件里，然后在页面的头部插入指向这个 js 文件的引用。

缓存失效的操作也很简单：删除这个文件就可以了。你可以通过运行一个周期性的任务，将 N 分钟前创建的文件都删除掉，来实现 TTL 失效策略。

如果想对缓存的尺寸做限制，那你可以实现一个最近最少使用（Least Recently Used, LRU）的失效策略，根据缓存内容的创建时间来删除内容。

这个失效策略需要你在文件系统的挂载 (Mount) 选项上开启“访问时间”这个开关项。(实际操作时忽略 noatime 挂载选项来达到这个目的)。如果这么做了, 你就应该使用内存文件系统来避免大量的磁盘操作。更多内容请查看第 331 页的“选择文件系统”。

10.3.3 缓存控制策略

Cache Control Policies

缓存引出的问题跟你数据库设计时违背了基本范式一样: 它们包含了重复数据, 这意味更新数据时要更新多个地方, 还要避免读到过期的“坏”数据。以下是几个常用的缓存控制策略:

存活时间

每个缓存的对象都带有一个作废日期, 用一个删除进程定时检查该数据的作废时间是否到达, 如果是就立即删除它, 你也可以暂时不理睬它, 直到下一次访问它时, 如果已经超过作废时间, 那才用一个更新的版本来替换它。这种作废策略最适用于很少变动或几乎不用刷新的数据。

显式作废

如果缓存里的数据过于“陈旧”而无法被接受, 那么更新缓存数据的进程就立即将该旧版本的数据作废。这个策略里有两个变体类型: 写一作废和写一更新。写一作废策略非常简单: 直接将该数据标志为作废 (也可以有从缓存里把它删除掉的选择)。写一更新策略就有更多的工作要做, 因为你还要用最新的数据来替换旧缓存数据。但是, 这个策略非常有用。特别是当生成缓存数据的代价很昂贵时 (这个功能在写的进程里可能已经具备)。更新了缓存之后, 将来的请求就用不着再等应用来生成这份数据了。如果你是在后台执行作废过程的, 比如是基于 TTL 的作废过程, 你可以在一个独立于任何用户请求的进程里生成最新版本的数据去替换缓存里已作废的数据。

读时作废

相对于在改变源数据时使缓存里对应的旧数据作废, 有一个替代性的方法是保存一些信息来帮你判断从缓存里读出的数据是否已经作废。它有个比显式作废更显著的优点: 随着时间的增长, 它开销是固定的。假设你要将一个对象作废, 而缓存里有 100 万个对象依赖于它。如果在写时将它作废, 你就不得不将缓存里的相关 100 万个对象都作废。而 100 万次读的延迟是相当小的, 这样就可以摊薄作废操作的时间成本, 避免了加载时的长时间延迟。

469 采用写时作废策略的最简单的方法是实行对象版本化管理。在这个方法里, 当把对象保存到缓存里时, 你同时要保存该数据所依赖的版本号或时间戳。举例来说, 假设你将一个用户在博客发表的文章的统计信息保存到缓存里, 这些信息包括了发表文章的数量。当将它作为 `blog_stats` 对象缓存时, 你同时也要把该用户当前的版本号也保存起来, 因为这个统计信息依赖于具体某个用户。

无论什么时候你更新了依赖于用户的数据, 也要随之改变用户的版本号。假设用户版本初始为 0, 你生成并缓存这些统计信息。当用户发表了一篇文章后, 你就将用户版本号改为 1 (最好将这个版本号与文章存放在一起, 尽管这个例子我们不必这么做)。那么, 当你需要显示统计信息时, 就先比较缓存的 `blog_stats` 对象的版本和缓存的用户版本, 因为这时用户的版本比这个对象的版本要高, 这样你就知道这份统计信息里的数据已经陈旧, 须要更新了。

这种用于内容作废的方法相当粗糙, 因为它预先假设了缓存里的依赖于用户的数据也跟其他数据进行互动。这个条件并不总是成立。举例来说, 如果用户编辑了一篇文章, 你也会去增加用户的版本号, 这使得缓存里的统

计数据都要作废了，哪怕真正的统计信息（文章的数目）实际上根本没发生变化。折中的方案是朴素的，一个简单的缓存作废策略不仅仅要易于实现，还要有更高的效率。

对象版本化管理是**标签式缓存**的一个简化形式，后者可以处理更复杂的依赖关系。一个标签化缓存了解不同类别的依赖关系，并能单独追踪每一个对象的版本号。在上一章的图书俱乐部的例子里，你可以这样给评论做缓存：用用户版本号和书本版本号一起给评论做标签，具体像 `user_ver=1234` 和 `book_ver=5678` 这样。如果其中一个版本发生了变化，你就要刷新缓存。

10.3.4 缓存对象的层次化

David Sifjan, *Memcached*

把对象按层次结构存放在缓存中，有助于读取、作废和内存使用的操作。你不仅要将对对象本身缓存起来，还要缓存它们的 ID 和对象分组的 ID，这样就能方便成组地读取它们。

电子商务网站上的搜索结果就是这种技术很好的例子。一次搜索可能返回一个匹配的产品清单，清单里包含了产品的名称、描述、缩略图和价格。如果把整个列表存放到缓存里，那读取时的效率是低下的，因为其他的搜索可能也会包含了同样的某几个产品，这样做的结果就是数据重复、浪费内存。这个策略也难以在产品价格发生变化时到缓存里找到对应的产品并使其作废，因为必须逐个清单地去查看是否存在这个价格变化了的产品。

一个可以代替缓存整个清单的方法是把搜索结果里尽量少的信息缓存起来，比如搜索的结果数目和结果清单里的产品 ID，这样你就可以单独缓存每一个产品资料了。这个方法解决了两个问题：一是消除了重复数据；二是更容易在单独产品的粒度上将缓存数据作废。

这个方法的缺点是你不得不从缓存里读取多个对象数据，而不是立即读取到整个搜索结果。然而，另一方面这也让你能更快地按照产品 ID 对搜索结果进行排序。现在，一次缓存命中就返回一个 ID 列表，如果缓存允许一次调用返回多个对象（Memcached 有一个 `mget()` 调用支持这个功能），你就可以用这些 ID 再到缓存里去读取对应的产品资料。

如果你使用不当，这个方法也会产生古怪的结果。假设你使用 TTL 策略来作废搜索结果，当产品资料发生变化时，明确地将缓存里对应的单个产品资料作废。现在试着想象一个产品的描述发生了变化，它不再包含跟缓存里搜索结果匹配的关键字，而搜索结果还没到作废时间。于是，你的用户就会看到“陈旧”的搜索结果，因为缓存里的这个搜索结果仍然引用了那个描述已经发生变化的产品。

对于多数应用来说，这一般不成为问题。如果你的应用无法容忍这个问题，那么就可以使用以版本为基础的缓存策略，在搜索之后，把产品版本号和搜索结果放在一起。在缓存里找到一个搜索结果后，把结果里的每个产品的版本号跟当前产品的版本号（也是在缓存里的）进行比较，如果发现有版本不符的，就通过重新搜索来获取新的搜索结果。

10.3.5 内容的预生成

Gregg Rahn, *Python*

除了应用层面上缓存数据之外，你还可以使用后台进程向服务器预先请求一些页面，然后将它们转换为静态页面保存在服务器上。如果页面是动态变化的，那你可以预生成页面中的一部分，然后使用一种技术，比如服务端整合，来生成最终页面。这样有助于减少预生成内容的大小和开销，因为本来你要为了各个最终页面上的

细微差别而不得不重复存储大量的内容。

缓存预生成的内容会占用大量空间，也不可能总是去预生成所有东西。无论哪种形式的缓存，预生成内容里的最重要部分就是请求最多的那些内容。因此，像我们在本章的前面提到过的那样，你可以通过 404 错误处理程序来对内容作“按需生成”。这些预生成的内容一般都放在内存文件系统里，避免放在磁盘上。

470 10.4 扩展 MySQL

Extending MySQL

如果 MySQL 完不成你所需要的任务，有一种可能性就是扩展它的能力。在这里，我们不是打算告诉你怎么做扩展，而是要提一下这个可能性里的一些具体途径。如果你有兴趣去深究其中的任何一条途径，那么网上有很多资源可供使用，也有很多关于这个主题的书可以参考。

当我们说“MySQL 完不成你所需要的任务”时，其中包含了两个含义：一是 MySQL 根本做不到，二是 MySQL 能做到，但是使用的办法不够好。无论哪个含义都是我们要扩展 MySQL 的理由。一个好消息是 MySQL 现在变得越来越模块化、多用途了。举例来说，MySQL 5.1 有大量可用的功能插件，它甚至允许存储引擎也是插件形式的，这样你就用不着把它们编译到 MySQL 服务器里了。

使用存储引擎将 MySQL 扩展为特定用途的数据库服务器是个伟大的想法。Brian Aker 已经编写了一个存储引擎的框架和一系列的文章、幻灯片来指导用户如何开发自己的存储引擎。这已经构成了一些主要的第三方存储引擎的基础。如果跟踪 MySQL 的内部邮件列表，你会发现现在有许多公司正在编写他们自己的内置存储引擎。举例来说，Friendster 使用一个特别的存储引擎来做社交图操作，另外，我们还知道有一家公司正在做一个用来做模糊搜索的引擎。编写一个简单的自定义引擎一点也不难。

你也可以把存储引擎直接用作软件某一部分的接口。Sphinx 就是个很好的例子，它直接与 Sphinx 全文检索软件通信（请查看附录 C）。

MySQL 5.1 也允许全文检索解析器插件，如果你能编写 UDF（请查看第 5 章），它擅长处理 CPU 密集的任务，这些任务必须在服务器线程环境下运行，对于 SQL 而言又太慢太笨重。因此，你可以用它们完成系统管理、服务集成、读取操作系统信息、调用 Web 服务、同步数据，以及其他更多相类似的任务。

MySQL 代理另外有一个很棒的选项，可以让你向 MySQL 协议增加你自己的功能。Paul McCullagh 的可扩展大二进制流框架项目 (<http://www.blobstreaming.org>) 为你打通了在 MySQL 里存储大型对象的道路。

因为 MySQL 是免费的、开源的软件，所以当你感觉它功能不够用时，你还可以去查看服务器代码。我们知道一些公司已经扩展了 MySQL 内部解析器的语法。近年来，还有第三方提交的许多有趣的 MySQL 扩展，涵盖了性能概要、扩展及其他新奇的应用。当人们想扩展 MySQL，MySQL 的开发者们总是反应积极，并乐于提供帮助。你可以通过邮件列表 internals@lists.mysql.com（注册用户请访问 <http://lists.mysql.com>）、MySQL 论坛和 IRC 频道 #mysql-dev 跟他们取得联系。

471 10.5 可替代的 MySQL

Alternatives to MySQL

MySQL 不是一个能适用于所有需要的万能解决方案。有些工作全部放到 MySQL 之外会更好，即使 MySQL 在

理论上也能做到。

一个很明显的例子是在传统的文件系统里对数据进行排序而不是在表里。图像文件是又一个经典的案例：你可以把它们都放在 BLOB 字段里，但是这在多数时候都不是个好主意（注 3）。通常的做法是把图像文件或其他大型二进制文件存在文件系统里，然后把文件名放在 MySQL 里。这样，应用就可以在 MySQL 之外读取文件了。在 Web 应用里，你可以把文件名放在元素的 src 属性里。

全文检索也是应该放在 MySQL 之外处理的任务之一——MySQL 不像 Lucene 或 Sphinx（请查看附录 3）那样擅长于这类检索。

NDB API 可以被用于某一类型的任务。比如，虽然 MySQL 的 NDB Cluster 存储引擎不适合在高性能要求的 Web 应用中作排序操作，但是可以通过直接使用 NDB API 来存储网站的 session 数据或用户注册信息。关于 NDB API，你可以访问 <http://dev.mysql.com/doc/ndbapi/en/index.html> 来获取更多信息。Apache 上也有相应的 NDB 模块，你可以从 <http://code.google.com/p/mod-ndb/> 下载。

最后，对于有些操作，比如图形化的关系、树的遍历，关系数据库并不擅长做这些。MySQL 也不擅长分布式数据处理，因为它缺少并行查询的执行能力。你可能需要使用别的工具（与 MySQL 一起使用）来达到这一目的。

注 3：使用 MySQL 复制功能能快速地将图像文件发布到其他机器上。据我们所知，一些应用使用了这项技术。

备份与还原

Backup and Recovery

人们很容易把重点放在“正经事”上，却忽视了备份和还原。实际上，紧迫的往往不是重要的，同样，重要的也未必显得很紧迫。备份在高性能应用里的重要性跟灾难还原一样，你需要从一开始就规划、设计好备份方案，这样在系统崩溃时，你就可以减少停机时间、性能缩水等负面影响。

如果事先没有规划、部署备份方案，那你就只能在事后创建“插入式”的解决方案了。在那个时候，你才发现之前在系统设计时做的一些决策已经堵住了处理高性能备份的最佳之路。举例来说，你已经架设好了一台服务器，然后又认识到需要一个 LVM 来做文件系统的快照——但为时已晚。你在为系统配置备份参数时，可能也不会注意到那几个会影响到性能的重要选项。同样，如果你没有规划和演练过系统还原预案，那到了真地要用它的时候，进展就不会顺利。

备份系统就像监控报警系统：许多系统管理员已经一次或多次重新“发明”过备份软件，这真非常可惜，因为市面上已经有许多质量上乘、支持完善、扩充性好的备份软件了——其中一些还是开源和免费的，我们鼓励你采用它们中对你有所帮助的那些。

在本章里，我们的内容不会覆盖一个设计良好的备份/还原方案的每一个环节。仅仅是这一章的标题就足够写出一本书——实际上，没几本书是专门讲这个问题的（注 1）。跟本书的第一版相反，我们假定许多读者在 MyISAM 之外还使用 InnoDB，或者用 InnoDB 代替 MyISAM。那种混合使用的状态使某些应用场景下的备份工作变得更复杂。

11.1 概况

Overview

在本章开始前，我们会先回顾几个术语，并讨论在规划自己的备份/还原方案，以及考虑未来潜在的需求时，你要记住的几个要点。然后，我们比较概括地讲述做备份时可供采用的技术，并在还原数据和灾难还原方面做一点探索。接着，我们会讨论如何选择一款适用于备份的工具。在文章的最后，我们用几个实例来讲解如何构建你自己的备份工具。

11.1.1 术语

Terminology

在开始之前，让我们澄清一些关键术语。首先，你将会经常听到热备份、温备份和冷备份这样的说法。人们常常用它们来形容备份对系统的影响，比如，“热”备份不需要任何一台服务器停机。问题是这三个术语对每个人

注 1：我们认为 W. Curtis Preston 的《Backup & Recovery》(O'Reilly)是个不错的选择。

来说并不是指同一件事情。有些工具甚至把“热”放在它们的名称里，但是实际上，它们做不到像它们名称所表示的那样热备份。我们会尽量避免使用这些术语，直接告诉你一个特定的技术或工具会对系统有多大的影响。

还有两个容易混淆的词：恢复和还原。本章中我们把它们用在特定的途径下。恢复的意思是从备份记录中获取数据，然后把它们加载到 MySQL 里，或者放在 MySQL 能找到的地方。还原在通常情况下的意思是描述援救整个系统或系统一部分的过程，其中就包含了将数据从备份记录中还原出来的操作，以及其他让系统恢复全部功能的必要步骤（比如重启 MySQL、更改配置、预热服务器缓存等）。

对许多人而言，还原仅仅指在服务器崩溃时修复损坏的数据表，不同于还原整个服务器。一个存储引擎的还原还要使数据和日志文件重新统一起来，它要确保只有生效的事务所作的数据库更新才能被写到数据文件里，然后再把日志文件里还未提交的事务重新提交。如果你使用的是事务性存储引擎，上面这个步骤就是整个还原过程的一部分了，甚至是制作备份记录的一部分。然而，这跟还原还不太一样，比如有时在意外的 DROP TABLE 之后，也需要这样来恢复数据。

11.1.2 所有关于还原的内容

It's All About Recovery

如果一切都运转正常，你就永远不需要考虑还原的事情。但是，当你需要还原时，那世界上最好的备份系统也帮不上忙，因为你需要的是一个很棒的还原系统。

但问题是使你的备份系统顺畅运转要比构建一个优良的还原过程和工具更加容易。以下就是具体原因：

- 先要有备份。如果没有事先的备份，你根本无法还原。所以在你构建一个系统的时候，你的注意力很自然地集中在备份上。因此，先对还原进行规划是相当重要的。实际上，在列出系统还原方面的需求之前，你就别想着构建备份系统。
- 备份是例行公事。这使你关注备份过程的自动化和调校，而没去考虑其他问题。在你的备份问题多盯上 5 分钟也不见得有多重要，但是在每一天里你关注过你的还原吗？你应该特意安排演练你的还原过程，直到它像你的备份过程一样顺畅、没有 Bug。
- 备份一般都不是在极大压力之下才做的，但是，还原往往发生在一个紧要关头，因此无论怎么强调还原的重要性都不会过分。
- 安全问题的妨碍。如果正在做离站式备份，你可能会去加密这些备份数据，或者用别的办法去保护它。这时，你一般把注意力放在当你的数据安全性受到危害之后会带来多少损失，而无视当需要还原这些数据却没有人能够解开你的加密卷时，带来的损失又是多少——或者当你需要从一个巨大的加密文件里解出那个文件时！
- 一个人就能规划、设计和实现备份系统，特别是有优秀的工具辅助时。当灾难来袭时，一个人可能就无法应付了。你需要培训好几个人来为系统还原做准备，这样就不会临时去找不合格的人来做数据还原了。

这里我们举一个取自真实世界的例子：一个客户报告说使用 mysqldump 做备份时，如果加上 -d 参数，备份速度就会快得像闪电一样，所以，他想知道为什么没有人提到用这个参数来加速备份过程的效果有这么大。如果这位客户试着恢复过这些备份记录，就会难以忘记其中的原因：-d 参数没导出数据！这位客户只关注备份，却没在意还原，所以完全没意识到这个问题。

当你开始考虑还原的时候，有个不错的主意是在你采取任何具体措施前先定义好你的需求。这里有几个因素需要考虑进去：

- 在产生严重后果前，你会丢失多少数据？你需要即时点（Point-In-Time）还原吗？或者说把最近一次正式备份以来的所有数据都丢失了，你能接受吗？有法律上的麻烦吗？
- 还原需要有多快？哪一种故障停工可以被你接受？哪种影响（比如局部不可用）对于你的应用和用户能够被接受？当上述场景发生时，你要构建怎么样的功能来维持系统的原有机能？
- 你确实需要还原吗？通常的需求是还原整个服务器、一个单独的数据、一张表、或者是一些指定的事务或语句。

475 针对这些问题，写下你的答案，并把它们加入系统文档里，在阅读本章的余下部分时，你要时刻牢记这些问题。先做这些练习可以帮助你注意力集中在还原上，如同规划你的备份方案时一样。把它们写在文档里是为了使你将来把这些步骤重新过一遍时更加方便。

备份神话 1：我把复制当备份用

这是我们经常遇到的一种误解。一个复制从服务器不是一个备份设备，RAID 阵列也不是。想知道为什么，就请这么来考虑：如果你在数据库上意外地执行了 DROP DATABASE 命令，你还能找回你的数据吗？RAID 和复制都通不过这个简单测试。它们不是备份，也不是备份的替代。只有备份才能满足备份的要求。

11.1.3 没有涉及的主题

Topics Not Worth Discussing

备份 MySQL 是众多普通的备份/还原方法中比较特殊的一种。我们想专注于高性能 MySQL，但是，不谈及一些其他的话题也会有一点困难，特别是因为我们看到过许多人都在同样的备份/还原问题里挣扎。以下是我们打算涉及的主题：

- 安全问题。（诸如谁能访问到备份记录、谁有恢复数据的权限、哪些文件需要加密。）
- 备份文件该存放在哪里，包括备份记录要离原始数据多远（在不同的磁盘上、不同的服务器上、或者是离站式的），怎么把数据从源移到目的地。
- 保留策略，审核，法律方面的要求及相关主题。
- 存储解决方案和存储媒体、压缩方案及增量备份。
- 存储格式（这里我们要多说一句：避免使用私有版权的备份格式）。
- 备份方案里的监控和报告功能。
- 内建于存储层的备份功能，或者像预制文件系统那样的特殊设备。

这些都是重要的主题。如果你对它们还不太熟悉，那最好去阅读一本关于备份的书籍。

11.1.4 大图景

The Big Picture

在开始引入所有可用选项的诸多细节前，先提供我们关于大多数人在构建一个备份/恢复系统时可能会需要的意见。你可以把这些建议作为一个出发点，或者工作方向的一个指引：

- 裸备份对大型数据而言是特别有必要的。它运行速度很快——这一点很重要。基于快照的备份是我们最喜欢选用的，如果使用的都是 InnoDB 表，那么 InnoDB 热备份就是很好的选择。
- 备份用于即时点还原的二进制日志。
- 多保留几个备份记录，二进制日志要足够长以便于你能从它们那里恢复出数据。
- 定期测试你的备份/还原过程，尤其是整个还原过程。
- 创建逻辑备份。（从效率考虑，可能还是基于你的裸备份来做较好。）
- 如果可能，要测试一下你的裸备份，以确保它们可以被用来做还原。在把它们复制到目的存储地址之前，即在备份的过程中测试它们。
- 对安全问题考虑周全。如果有人危及服务器安全时，他能获得访问备份服务器的权限吗？反之亦然。
- 使用备份工具来监控备份记录和备份过程。你需要外部工具来核实备份操作是没问题的。
- 灵活运用机器间复制文件的方法，有许多复制方法比 scp 和 rsync 更高效。更多内容请查看附录 A。

476

11.1.5 为什么要备份

Why Backup?

如果你正在构建一个依赖于 MySQL 的高性能系统，备份是很重要的。以下就是它的原因：

灾难还原

当你遭遇到硬件崩溃、恶意 Bug 破坏你的数据，或者服务器或数据因为某个其他的原因（这种可能的原因会很多，而且不尽相同——发挥你的想象力去想吧）不能继续使用的时候，你就需要做灾难还原了。以上任何一种灾难发生的可能性相当低，而且，叠加起来一起发生的可能性更低。但是你要为以下这些可能做好准备：某个人碰巧错连到了你的服务器，然后执行了 ALTER TABLE 命令（注 2）；一把火烧毁了你的机房；对你的服务器发动了恶意攻击；MySQL 的一个 Bug。

人们改变了他们的想法

你会很惊奇地发现我们会经常需要把部分数据还原到它们在以前某个时间点上存在过的状态。对于一些应用而言，这样的情况的发生频率要比灾难高很多（比如说一个重要客户以前删除了一些数据，现在又想把它们还原回来）。

477

注 2：Baron 仍然记得当他还在一个电子商务网站做开发人员时，他在另外一个命令窗口里输入了这样一条命令，结果……这件事情也是 DBA 团队的过失，因为他们不该把当前服务器上的这个修改权限开放给开发人员使用。

审查

有时你需要知道数据或式样在过去的某个点上看起来是什么样子的。举例来说,你可能被卷入了一场官司,或者你在应用里发现了一个 Bug,现在想看看这段代码在过去运行时产生了什么样的结果(有时仅仅把代码放入版本控制软件还是不够的)。

测试

用真实数据做测试的最简便的方法是定时地把最新的产品数据刷新到测试服务器上。如果你已经做了备份,那就更简单了——使用备份数据就可以。

请检查你所有关于备份的假设。举例来说,你是不是假设共享主机提供商在用你的账号备份 MySQL 服务器?虽然共享主机跟高性能没什么关系,但我们想指出这样一个假设是靠不住的。(为了免得你再怀疑,我们可以说许多主机提供商根本不备份 MySQL 服务器,其他几家也只是在服务器运行的时候,做个数据库文件拷贝而已,这样的数据库文件拷贝很可能是损坏的,没法用的。)

11.2 要权衡的事项

备份 MySQL 要比看上去更难。从最基本的层面看,一个备份记录就是一份数据的副本,但是,诸如你应用所需的、MySQL 存储引擎架构、系统配置等因素使它操作起来变得困难。

11.2.1 你能承受的损失有多大

知道你能承担多大的数据损失可以用来指导你的备份策略。你需要一个即时点还原功能吗?或者说它是否可以被用作昨晚的数据备份,而不管期间丢失的那些数据?如果你需要即时快照式的还原,那你可能需要做常规备份,确保日志功能已经打开,这样,你就可以通过这个日志来恢复备份数据,并将它们还原到你需要的时间点上。

通常而言,你能承担的损失越大,备份就越容易做。如果你有严格要求,那就很难确保能还原所有的东西。不同的即时点还原有着不同的需求:“软”即时点还原的需求意味着你最好能重新创建你的数据,只有这样,数据状态才能跟问题发生时的那个时间点“靠得够近”。“硬”即时点还原的需求意味着你无法容忍任何已提交事务的丢失,不管何种糟糕事情的发生(比如服务器着火了)。这就需要一些特殊的技术,比如说把二进制日志放在一个单独 SAN 卷里,或者使用 DRBD 磁盘复制。更多内容请查看第 9 章。

11.2.2 在线备份还是离线备份

如果可以做到,停掉 MySQL 后再做备份是最简单、最安全的方法,而且还能在最小的“脏数据”和源-备份矛盾的风险下,获得与源数据库相一致的备份记录。如果你停掉 MySQL,你就可以放心去复制数据,不会碰到类似 InnoDB 缓冲池里的“脏缓存”之类的难题,也不用担心复制数据时这些数据被更新了。而且,这时系统没有来自应用的负荷,备份的速度因此会更快。

然而,让一个服务器离线的成本要比看起来高得多。即使把停机时间减到最小,关闭和启动高负载大数据量的

MySQL 也要耗费很长的一段时间：

- 如果在 InnoDB 的缓冲池里有很多“脏缓存”——它的意思是那些数据在内存里已经被更新了但是还没写到磁盘上去——InnoDB 要过很长的一段时间才把这些更新了的数据写到磁盘上。你可以使用 `innodb_fast_shutdown` 配置变量来影响 InnoDB 的关闭时间，这个变量可以控制 InnoDB 处理缓冲池的方式，让它在关闭之前把缓存更新到磁盘上（注 3）。这个方法并不能从根本上解决问题，只是把问题转移了。你仍然不能显著地减少 MySQL 的关闭-开启时间。有时，你可以通过在别的方面配置 InnoDB 来改善这个情况，但这些更改会对性能产生更大的影响。更详细的内容请查看第 281 页的“MySQL I/O 调优”。
- 重启也会消耗很长的时间。打开所有的表、预热缓存都是缓慢的过程，特别是表及其数据很多的时候。如果你设置了 `innodb_fast_shutdown=2` 来加快 InnoDB 的关闭速度，那在下次运行之初，InnoDB 就不得不执行一次还原过程。即使在你的服务器看似已经全部启动好了，它也要花费很长的时间来预热和做一些其他准备工作。

因此，如果你要求高性能构建你的应用，你就必须仔细设计你的备份方案，让你的产品服务器无需离线也能备份。从一致性需求来看，纵然备份可以在线进行，仍然意味着它会很明显地影响原来的系统服务。

举例来说，在许多备份方法里最常见的示例之一就是 `FLUSH TABLES WITH READ LOCK` 开始的。这段代码告诉 MySQL 去刷新（注 4）所有表的缓存然后锁定，同时刷新查询缓存。

这个需要好一会儿才能完成。（很难确切地说需要多长时间。如果全局的读锁不得不等待一个很长的语句执行结束，或者你有许多表，这个时间都会变得更长。）在锁被释放之前，你无法修改服务器上的数据，跟服务器关机相比，`FLUSH TABLES WITH READ LOCK` 的代价也不算昂贵，因为大多数缓存还放在内存里，服务器还是“温热”，但是这个方法相对来说还是带有破坏性的。

如果这构成了问题，那你就需要找到一个可替代的方法。我们使用的一个方法是在复制从服务器上做一个备份，这台服务器是从服务器池里的一台，所以离线-上线的成本很低。让我们回到本章的主题，其他关于在线和离线备份的内容，在本章后面会讲到。现在，我们只能这么说：做在线备份，同时又不影响到服务器上的服务，这很难做到。

11.2.3 逻辑备份还是裸备份

备份 MySQL 数据有两种方法：做一个逻辑备份（也叫做“导出”）和复制裸文件。一个逻辑备份里的数据是 MySQL 能识别的格式，它或者是 SQL，或者是带分界符号的文本（注 5）。裸文件就是原来存储在磁盘上的那些文件。

两种复制数据方法里的每一种都有各自的优缺点。

注 3：插入操作的缓冲区存储在 InnoDB 的表空间文件里，与其他数据放在一起；一个后台进程最终会把这些插入的记录归并到它们对应的表里。

注 4：刷新 MyISAM（不是 InnoDB）里的缓存到磁盘。

注 5：由 `mysqldump` 生成的逻辑备份不总是文本文件。从 SQL 导出的数据包含了许多种不同的字符集，甚至还会包含无法作为可见字符显示的二进制数据。而且，对于许多编辑器而言，它们的行也可能太长。尽管如此，许多备份文件还是可以被文本编辑器打开并阅读的，特别是当你使用带有 `-hex-blob` 选项的 `mysqldump` 生成备份时。

逻辑备份

逻辑备份有以下这些优点：

- 它们都是普通的文件，可以使用编辑器或命令行工具（比如 `grep`、`sed`）来操作和检查。这在恢复数据时很有用，在只是检查一下数据而不做恢复时也一样用得上。
- 它们很容易被恢复。你可以利用管道（Pipe）把它们传入 `mysql` 或使用 `mysqlimport` 导入。
- 可以跨网络进行备份和恢复——不在这台 MySQL 主机上，而在其他机器上进行。
- 它们有很强的兼容性，因为 `mysqldump`——许多人都喜欢采用这个工具（注 6）——能接受许多选项，比如可以用 `where` 语句来限制哪些数据行需要备份。
- 它们独立于具体的存储引擎。因为你是从 MySQL 服务器上将它们取出来的，已经对底层数据引擎的差异做了抽象。因而，只需少许工作就能把从 InnoDB 里备份出来的表恢复到 MyISAM 表里。这在裸文件副本里是做不到的。
- 在许多情况下，如果你使用 `mysqldump` 时指定了正确的选项，你甚至可以把你的逻辑备份记录导入别的数据库服务器，比如 PostgreSQL。
- 它们可以避免数据损坏。如果你复制裸文件时，磁盘驱动器突然坏掉了，你就只能得到一个损坏了的数据备份。除非你特地检查一下这个备份，否则不会注意到这一点，直到将来要用到它的时候。如果这些 MySQL 数据的内存还是完好的，那么，逻辑备份有时还可以得到一个可信赖的备份，而一个完好的裸文件副本则已经是不可能的。

逻辑备份也有如下这些缺点：

- 服务器必须亲自来生成它们，所以会使用更多的 CPU 周期。
- 逻辑备份在某些时候会比原来数据文件更大（注 7）。用 ASCII 表示的数据在效率上有时不及存储引擎存储数据时使用的格式。举例来说，一个整数在存储里是 4 个字节，写成 ASCII 码后，它最多的时候需要 12 个字节。你可以有效地使用压缩功能，但这会使用更多的 CPU 资源。
- 浮点数会因此丢失精度信息，这妨碍了备份记录中数据的精确恢复。（Google 对 MySQL 的一些补丁里就包括了一个针对 `mysqldump` 中此类精度问题的。）
- 从逻辑备份里恢复出数据需要 MySQL 加载和解释这些声明，并重建索引，这将给服务器增添更多的工作量。

这里面最大的不利因素是从 MySQL 导出数据的开销和使用 SQL 语言把数据导回去的开销。

裸备份

裸备份有下面这些优点：

- 裸文件备份仅仅要求你把那些文件复制到其他地方，就算备份了。裸文件不要你做任何额外的工作来生成。
- 恢复裸备份的数据更加简单，不管是基于哪个存储引擎。对于 MyISAM，就是把文件复制到指定的位置。

注 6：也有其他可选择的，有一类工具它们能并行导出和恢复数据，但这个是最常用的工具。

注 7：在我们的经验里，逻辑备份文件往往比裸备份的小，但它们不总是这样的。

对于 InnoDB，它要求你先关闭 MySQL 服务器，然后再完成余下的步骤。

- 裸备份往往便于跨平台、操作系统和 MySQL 版本。
- 恢复裸备份里的数据的速度更快一些，因为 MySQL 不需要执行任何 SQL 语句，也不需要重建索引。如果备份里包含了无法整个放入服务器内存的 InnoDB 数据表，那恢复数据更会快得多。

以下是裸备份的一些不利之处：

- InnoDB 的裸文件经常要比相应的逻辑备份大很多。一般在 InnoDB 的表空间都保留了大量闲置的空间，这里面相当多的是用于存储数据之外的用途（比如插入操作的缓冲区、回滚段等）。
- 裸备份也不总是可以轻易地跨越平台、操作系统和 MySQL 版本。文件名的大小写敏感和浮点数格式都会成为麻烦的根源。你不能把这些备份移到浮点数格式不一样的服务器上去恢复（但是，目前大量的主流处理器都使用 IEEE 的浮点数格式）。

裸备份通常更容易做，也更有效率，但是你不能把它当作长期手段或合理的需求。你至少要定期地做一次逻辑备份。

在你测试之前，不要想当然地认为这个备份（特别是裸备份）是好用的。对于 InnoDB 而言，这意味着要开启 MySQL 实例，然后运行 InnoDB 还原，再运行 CHECK TABLES。你也可以跳过这一步，直接在备份文件上运行 innochecksum——但是，我们不建议这么做。对 MyISAM 而言，你应该运行 CHECK TABLES，或者使用 myisamchk。

一个比较灵活的方法是采用这两种备份混合方式：先做一个裸副本，然后开启一个 MySQL 服务器，在这些副本基础上生成逻辑备份。这样你就获得两者的优点，不会使服务器在导出时被加上过重的负担。当你有文件系统快照的功能时，这个方法会显得特别方便——你可以做一个快照，然后把快照复制到另外一台服务器上释放出来，测试这些裸文件，最后做一个逻辑备份。

11.2.4 要备份什么

What to Back Up

你的还原需求会告诉你要备份哪些东西。最简单的策略是只备份数据和表定义，这算是最最少的备份内容了。通常来讲，为了能还原服务器，你需要备份大量的更多的东西。以下这些是在你做备份时应考虑的备份内容：

不显眼的数据库

不要忘记那些容易被忽视的数据，诸如二进制日志、InnoDB 事务日志。

代码

现代 MySQL 服务器能存储大量代码，比如触发器、存储过程。如果备份了 mysql 的数据库，那你也需要备份这些代码。但是，之后就很难以此还原出一个完整的数据库，因为在那个数据库里的某些“数据”——就像存储过程之类的，实际上只保存在 mysql 数据库里面。

复制的配置信息

如果你正在恢复一台带有复制的服务器，你也要把你需要的所有复制文件备份过来，比如二进制日志、转发日志、日志索引文件及 .info 文件。在最低要求上，你应该包含 SHOW MASTER STATUS 和/或 SHOW SLAVE STATUS 的所有输出内容。使用 FLUSH LOGS 使 MySQL 开启一个全新的日志文件也很有用处，在做即时点

还原时，从日志文件起点开始还原要比从中间开始更加容易。

服务器的配置信息

如果你要将服务器从一个现实世界的灾难还原过来——比如说地震过后，你正在重建数据中心里的一台受损的服务器——这时，你看到备份数据里的服务器配置文件会万分感激。

被选中的操作系统文件

和服务器配置文件一样，备份那些对服务器来说必需的外部配置文件也很重要。在 Unix 服务器上，这些文件包括 cron 任务、用户和组的配置、管理脚本及 sudo 规则。

这些建议在许多场景里会被很快地翻译为“备份所有的东西”。如果你有大量的数据，这么做的代价会变得昂贵，在关于怎么做备份这个问题上，你必须有个灵活的方法。特别地，你可能会希望把不同的数据备份到不同的备份记录里。举例来说，你可以把数据、二进制日志、操作系统及其配置文件都单独备份。

增量备份

对于大量的数据，一种常用的办法就是定期做增量备份。下面是有关这种备份方法的进一步说明：

- 备份二进制日志。这是最简单、使用最广泛、最适合的增量备份的途径。
- 不要备份没有变动的数据表。有一些存储引擎，比如 MyISAM，对每张表的最后一次更新时间都有记录。你可以通过查看磁盘上的文件日期，或者运行 `SHOW TABLESTATUS` 来得到这个更新时间。如果你使用的是 InnoDB，那么就创建一个触发器，把这些更新时间都写入一个很小的“最近更新时间”表里。这个方法只能用在更新频率不高的那些表上，这样开销会很低。一个自定义的备份脚本能轻松地判定哪些表的数据有过变动。

如果你有“查找”表，其中包含的数据都是诸如用不同语言表示的所有月份的名称，或者是州名或地区名的缩写。这样的表最好是放在一个单独的数据库里，这样你就用不着总是去备份它们。

- 不要备份没发生改变的行。如果一个表是只能 INSERT 的，比如记录每个页面的点击数的表，你就往里面增加一个 `TIMESTAMP` 字段，每一次备份自上次备份以来新增的那些行就可以了。你也可以使用 Merge 数据引擎，把老的数据放在静态表里。
- 有些数据不要去备份。有时候这个意义重大，举例来说，你有一个数据仓库，它是基于其他数据构建的，因此从技术方面来讲这也属于冗余。在备份时，你可以只备份这部分原始数据，用不着把数据仓库也备份起来。即使在还原数据后需要花时间去重建数据仓库，这个方法也是可行的，因为在备份时候省下的时间和存储空间，会多于全部备份时所需的时间和存储空间。你也可以选择不备份某些临时数据，比如存储网站 session 数据的表。
- 只备份二进制日志里的变化部分。你可以使用 `rdiff` 命令只获取自日志上次备份后变化的记录（定期的完全备份还是需要的）。还有一个我们用到的工具是 `backup`，它把 `rdiff` 和 `rsync` 合并为一个完整的备份方案。或者你可以在每次备份后都用 `FLUSH LOGS` 来重新创建一个日志，这样就无需记录日志的变化了。
- 只备份数据文件里的变化部分。这个就像刚才处理二进制日志一样。在 Unix 上常用的工具是 `rdiff` 和 `rdiffbackup`。这种方法对于数据量很大但变化不大的数据库非常有用。假如你有 1TB 的数据，而其中只有 50GB 每天会有变动的，那么，现在只要备份其中每天发生变动的那部分就可以了，再偶尔做一次完全备份。这样做的好处是你可以把变动部分转换为一系列磁盘读/写的方式，写入一个完全备份的记录，

比使用二进制日志要快很多。然而，做一次二进制差异备份可能要比完全备份慢一些。

增量备份的缺点是数据还原时的复杂性提高了。如果你正顶着重重压力做系统恢复，相比于还原一个接一个的增量备份，还原一个备份记录会使你轻松许多。如果能做完全备份，那么为简单起见，我们还是建议您做完全备份。

无论如何，你偶尔还是要做一次完全备份的——建议您至少一星期一次。我们不希望使用一个年度的所有增量备份来恢复系统，甚至连一个星期都包含了很大的工作量和风险。

11.2.5 存储引擎和一致性

Storage Engines and Consistency

MySQL 能够选择使用各种存储引擎，这让备份操作变得更加复杂。事情的关键是如何在不同的存储引擎上获得一个与服务器一致的备份记录。

这里的一致性要从两个方面来考虑：**数据一致性和文件一致性**。

数据一致性

484

在备份的时候，你必须保证数据在即时点上是一致的。举例来说，在一个电子商务的数据库里，你需要确保发票和支付记录是相互一致的。当恢复一个支付记录时，若没有其对应的发票，或者反过来，这两个情形都会引起麻烦。

如果你正在做在线备份（在一台正在运行的服务器上），你需要确保获得的是与全部相关表都一致的备份。这就意味着你不能是一次一个表地做锁定-备份的操作。如果你没有使用事务性数据存储引擎，你就别无选择，只有使用 `LOCK TABLES` 命令锁定所有的表，然后把它们备份到一起，直到所有相关的表都备份完了，你才可以释放锁。

InnoDB 的 MVCC 功能可以帮上点忙。你可以先开启一个事务，然后导出一组相关表，最后提交事务。（这里你使用一个事务来获得一致的备份，就不用 `LOCKTABLES` 命令了，因为事务会被隐式提交。——细节方面请查看 MySQL 帮助手册）只要你使用的是 `REPEATABLE_READ` 事务隔离水平，这样的做法就能给你一个完美的一致性。数据的即时点快照也不会阻碍服务器备份完成后的进一步的工作。

然而，这个方法也不能让你避开设计糟糕的应用逻辑。假如你的电子商务站点插入了一条支付记录，然后提交了事务，接着又另外开启一个事务，插入了一条发票记录。你的备份过程可能正好是在这两个事务之间开始的，于是，备份里只有这条支付记录而没有发票记录了。所以说你必须仔细地设计你的事务，把相关的操作都组织起来放入一个事务里。

你也可以使用 `mysqldump` 来获得逻辑上一致的 InnoDB 表的备份，备份时给 `mysqldump` 加上 `--single-transaction` 选项就可以获得上面讲到过的效果了。但是，这会产生一个很长的事务，因此，在某些时候甚至会产生大到难以接受的系统开销。

有一些工具支持“备份集”，比如 ZRM（下文中会讲到）、Maatkit 的 `mk-parallel-dump`，它们可以帮你轻松备份相关的表集合。

文件一致性

内部文件相互一致也非常重要，比如备份记录反映不出一个文件是否正处于 UPDATE 过程的中间。备份的文件之间相互也要保持一致性，否则当你恢复这些文件时会遇到让你感觉糟糕的意外（这些文件可能都被损坏了）。如果你在不同的时间里复制了这些相关连的文件，那它们就无法一致了。MyISAM 的 .MYD 和 .MYI 就是一个例子。

485 如果使用的是一个非事务类型的数据引擎，例如 MyISAM，你唯一可选择的就是锁定表然后刷新缓冲区。这就意味着你或者使用 LOCK TABLES 和 FLUSH TABLES 的组合，把内存里缓存着的更新写到磁盘上，或者使用 FLUSH TABLES WITH READ LOCK。一旦缓存区刷新后，你就能安全地取得 MyISAM 文件的一份裸副本。

如果使用的是 InnoDB，想要确定磁盘上的文件是否一致就有点困难了。即使你做了 FLUSH TABLES WITH READ LOCK 操作，InnoDB 仍然会在后台运行着：它的插入缓冲区、日志和写线程继续把数据更新写到日志和表空间的文件里。这些线程都被设计成异步的——这样的设计是为了 InnoDB 能达到高并发性——所以，它们独立于 LOCK TABLES。因而，你不仅要确保每个文件的内部一致性，还要在同一时刻复制日志文件和表空间文件。如果做备份时，刚好有个线程在修改文件，或者说备份日志文件的时间点跟备份表空间文件时不一样，那在还原这些文件后得到的还是一个数据损坏了的系统。不过，你可以通过两条途径来避免这个问题：

- 一直等待，直到 InnoDB 里那些清除线程和插入缓冲区合并线程退出。你可以查看 SHOW INNODB STATUS 的输出结果，当看到缓冲区都干净了，未决的写操作也都完成了，你就可以开始复制文件了。然而，这个方法实行起来会花费很长一段时间，而且由于 InnoDB 后台线程的存在，操作时会包含很多猜测，从而显得不怎么可靠。有鉴于此，我们不推荐这个方法。
- 利用系统功能，比如用 LVM 对数据和日志文件做一个一致性快照。这个快照必须使两者成对进行，如果分开来做快照那就不怎么好了。我们将在本章的后面详细讨论 LVM 快照。

一旦把文件复制到了别处，你就可以释放所有的锁，让 MySQL 继续正常运行。

11.2.6 复制

Replication

在普通大众的观念里，用 MySQL 复制做备份的想法是荒谬的。将复制用作完全备份的一部分确实有其特有的优点，但是不能把它当做备份的一切——通常就是这么说的。

从一个从服务器上取得备份的最大好处是它不会妨碍主服务器的运行，也不会给它增加额外的负载。这就是建立一个从服务器的好理由，哪怕你不是拿它来做负载平衡或高可用性。如果不舍得花钱，你也可以把这台备用用途的从服务器派上别的用场，比如做数据报告——只要这些任务不写入数据，也不更改你正在备份的数据。这台从服务器无需专门用作备份，它只要在下次备份到来前，与主服务器的数据同步更新就可以了。

486 当你在一台从服务器上做备份时，要保存所有关于复制过程的信息，比如这台从服务器在主服务器的地位。这个对于以下几种情况很有用：新从服务器的克隆、把二进制日志导入主服务器做即时点还原、把一个从服务器升级为主服务器，以及其他更多的情形。在你停止从服务器之前，要确保没有被打开的临时表，因为它们会阻碍你重启复制功能。关于这方面的更多内容你可以查看第 394 页的“丢失的临时表”。

如果在从服务器里有一台特意设置了延迟复制，那它在一些灾难场景后做还原数据时会很有用。假设你是延迟一小时做复制的，当一条不必要的语句在主服务器上执行之后，你还有一小时的时间去注意到它，并能在从服

务器重复这个操作之前，停止其复制行为。然后，你就把这台从服务器升级为主服务器，跳过刚才那个有害的语句，把一些数量较小的相关日志事件重放一遍就好了。这比接下来我们要讨论的即时点还原技术更加快速。来自 Maatkit 的 `mk-slave-delay` 脚本可以用在此处。



警告：从服务器的数据可能会跟主服务器的不一样。许多人都假定从服务器上的数据就是主服务器上数据的完全一致的副本。但是，根据我们经验，数据错配在从服务器上很常见，MySQL 也没办法检测到这个问题。备份了错误的或损坏的数据，结果都是得到一个没用的备份记录。更多关于如何确保从服务器数据跟主服务器一致的内容请查看第 380 页的“确定从服务器与主服务器数据一致”。第 8 章也提到了一些避免主-从服务器之间不一致的建议。

保存一份数据复制副本，可以帮你远离例如主机磁盘崩溃等问题，但这也不能保证它一直有效，毕竟，复制不是备份。

11.3 管理和备份二进制日志

Managing and Backing Up Binary Logs

服务器上的二进制日志是你要备份的最重要的内容之一。它们对于即时点还原是必不可少的，因为在尺寸上它比数据小，易于经常备份。如果你有某个时间点上的数据备份，还有自那个时间点以来的所有二进制日志，那么你就可以通过重放二进制日志，把自上次完全备份以来所有的数据变动都“前滚”出来。

MySQL 也把二进制日志用作复制。这意味着你的备份/还原策略要经常与你的复制配置相配合。

二进制日志很“特殊”。如果你丢失了数据，又不甘心它们就这么没了，为了让此类事件发生概率降到最小，你可以把它们放在一个单独的卷里。这样即使你想用 LVM 对它做一个快照，也是没问题的。如果想要更多的安全保障，你可以把它们放在 SAN 里，或者通过 DRBD 复制到其他设备上。关于这方面的更多内容，你可以阅读第 9 章。

经常备份二进制日志是个很好的主意。如果你无法承受超过 30 分钟的数据被丢失，那你就该至少每 30 分钟备份一次日志。你也可以使用带 `--log_slave_updates` 选项的只读复制从服务器，以取得额外一层的安全。这样，虽然日志的位置就与主服务器的不匹配，但是，要找到还原的正确位置也不是太难。

以下是我们推荐的用于二进制日志的服务器配置：

```
log_bin                = mysql-bin
sync_binlog            = 1
innodb_support_xa      = 1    # MySQL 5.0 and newer only
innodb_safe_binlog     # MySQL 4.1 only, roughly equivalent to innodb_support_xa
```

此外，还有别的配置项可用，比如限制日志大小的选项。这方面的更多内容你可以查看 MySQL 的使用手册。

11.3.1 二进制日志的格式

The Binary Log Format

二进制日志里包含了一系列的顺序事件。每个事件都有固定大小的头部信息，其中包含着多种信息，比如当前的时间戳、默认的数据库等。你可以使用 `mysqlbinlog` 工具来查看日志的内容，它能打印出一些头部信息，以下就是一个输出的例子：

```

1 # at 277
2 #071030 10:47:21 server id 3 end_log_pos 369 Query thread_id=13 exec_time=0
  error_code=0
3 SET TIMESTAMP=1193755641/*!*/;
4 insert into test(a) values(2)/*!*/;

```

第 1 行包含了当前记录在日志文件里的偏移字节数（这里是 277）。

第 2 行包含了如下内容：

- 事件发生的日期和时间，MySQL 也会用这个来生成 SET TIMESTAMP 语句。
- 源服务器的服务器 ID，这个对于防止循环复制及其他问题是必需的。
- end_log_pos 是描述下一个事件的偏移字节数。这个值在多语句事务环境下的多数事件记录里都是累加的。在主服务器上发生这些事务时，MySQL 会把这些事件都复制到一个缓存里，但是并不知道下一个日志事件的位置。
- 事件的类型。在本例中，它是一个查询。其实还存在许多不同的类型。
- 在服务器上执行该事件的线程的 ID，跟执行 CONNECTION_ID() 函数一样，这在审核时很重要。
- exec_time，这个项目的真实含义即使对于一些 MySQL 开发人员来说也不太明白。通常来讲，它记录了这个语句执行时所花费的时间，但是，在某些条件下，它又会是些奇怪的值。举例来说，当一个从服务器的 I/O 线程远远落后于主服务器时，这个项目在转发日志上会是一个很大的数值，不管这些语句在主服务器上执行得有多快。所以，不要去依赖这个项目的值。
- 该事件在源服务器上产生的错误代码。如果当从服务器上重放该事件时产生了跟它不一样的错误代码时，作为一项安全预防措施，复制过程就会失败。

其他行里还包含了 SQL 要重放该事件所需的其他数据。还有用户自定义变量、任何其他特殊的设置，例如语句执行时受影响的时间戳也会显示在这里。



提示：如果你正在使用 MySQL 5.1 里的以行为基础的日志，这些事件就不是 SQL 的表现形式了，而是一种无人能读的表更新语句的“映像”。

11.3.2 安全地清除旧日志

Handling Old Binary Logs Safely

现在，你需要制定一个日志作废策略，以免 MySQL 用二进制日志把磁盘都填满了。日志大小增速与 MySQL 的工作量和日志格式有着相互联系（MySQL 5.1 里以行为基础的日志，会有更多的日志条目）。我们建议您的日志只要是有可能被用到，就应该保存下来。这些保存下来的日志可以帮你搭建复制从服务器、分析服务器的工作负荷、审核，以及做上次完全备份以来的即时点还原。当你要在决定该保留多长的日志时，请考虑一下以上那几个需求。

一个常用的设置方法是使用 expire_logs_days 变量来告诉 MySQL 要过多久才可以清除这些日志。这个参数只能用于 MySQL 4.1 以后的版本。对于以前的版本，你就不得不手工删除。然而，你可能也看到过那个建议：使用 cron 条目去删除旧日志，就像下面这样：

```
0 0 * * * /usr/bin/find /var/log/mysql -mtime +N -name "mysql-bin.[0-9]*" | xargs rm
```

对于早于 4.1 版本的 MySQL 来说，这是清除日志的唯一手段了，千万不要在 4.1 及更新的版本里这么做。使用

rm 删除日志文件会使 mysql_bin.index 状态文件跟磁盘上的文件不同步，像 SHOW MASTER LOGS 命令执行起来会毫无反应。即使再手工去修改 mysql_bin.index 文件也无济于事。只能像下面这样使用 cron 命令去做：

```
0 0 * * * /usr/bin/mysql -e "PURGE MASTER LOGS BEFORE CURRENT_DATE - INTERVAL N DAY"
```

expire_logs_days 的设置值在服务器重启，或者 MySQL 轮转日志之后才能生效，所以，如果二进制日志还没填满，也没轮转，服务器仍然不会去清除那些旧的日志条目，它是根据条目的更新时间而不是内容做清除操作的。

11.4 数据备份

在很多主题里，要做个备份总有很多途径，或好或坏——显而易见的方法有时不见得是个好办法。这样的方法往往通过夸大网络、磁盘和 CPU 的承载能力，把备份的速度说得尽可能地快。其实这些因素之间是有一个平衡点的，你要做几次试验才能找到这个“最近听音位置”。

我们很难给出一个特定的建议，所以，在此展示几种很常用的技术。

11.4.1 做一个逻辑备份

关于逻辑备份，首先要认识到的是它们创建方式是不一样的。在实际应用中，有两种逻辑备份：SQL 导出和限定符文件。

SQL 导出

SQL 导出是很多人所熟悉的，因为这是 mysqldump 默认创建的。举例来说，用默认设置导出一个小的数据表，其输出结果如下所示（已作删减）：

```
$ mysqldump test t1
-- [Version and host comments]

/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
-- [More version-specific comments to save options for restore]

--
-- Table structure for table `t1`
--

DROP TABLE IF EXISTS `t1`;
CREATE TABLE `t1` (
  `a` int(11) NOT NULL,
  PRIMARY KEY (`a`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

--
-- Dumping data for table `t1`
--

LOCK TABLES `t1` WRITE;
/*!40000 ALTER TABLE `t1` DISABLE KEYS */;
INSERT INTO `t1` VALUES (1);
```



```

/*!40000 ALTER TABLE `t1` ENABLE KEYS */;
UNLOCK TABLES;
/*!40103 SET TIME_ZONE=@OLD_TIME_ZONE */;

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
-- [More option restoration]

```

这个导出的文件里包含了表的结构和数据，所有这些都是用可用的 SQL 语句的方式表示的。文件开始的注释部分用来设置 MySQL 各项参数，它们的作用或者是为了还原工作能进行得更有效率，或者是为了兼容性和正确性。接下来，你会看到表的结构，然后就是表内的数据了。最后的那段脚本是将文件开始时修改的参数再修改回来。

导出文件可以直接用来做还原操作，这样显得很方便，但是仅凭 mysqldump 的默认选项还是无法做大规模备份的（在下文里我们会深入讨论 mysqldump 的各个选项）。

mysqldump 不是做 SQL 逻辑备份的唯一工具，例如你也可以使用 phpMyAdmin 来创建备份。我们真正想指出的是使用任何一个特定的工具都不会有太多的问题，但是这里我们把做单一备份记录时的所有缺点都放在了显眼位置，以下就是主要的问题：

样式与数据存放在一起

如果想从单一文件恢复，这是很方便的，但是，如果只恢复一个表，或者只恢复数据，那做起来就麻烦了。导出两次——一次是数据，另一次是样式——可以减轻这个痛苦，但仍然会有下面这个问题。

巨大的 SQL 语句

对于服务器来说，解析、执行所有的 SQL 语句包含了巨大的工作量。通过这样的途径加载数据会相对显得缓慢。

单一的巨大文件

许多文本编辑器无法编辑大文件，在文件的行很长时也无法正常工作。虽然有时你可以使用命令行流编辑器——例如 sed 或 grep——取出你所需要那些数据，但是，与此相比，保持文件小型化才是更好的做法。

逻辑备份成本昂贵

同样也是把数据从 MySQL 里取出来，有很多途径比通过 client/server 协议发送结果集更有效率。

这些限制条件意味着当表变得很大时，SQL 导出的方式就无用武之地了。这里就有了另外一个选择：把数据导出到带定界符号的文件里。

使用定界符文件备份

你可以使用 SELECT INTO OUTFILE SQL 命令来创建一个定界符文件形式的逻辑备份。（你也可以使用带-tab 选项的 mysqldump 来生成，它会帮你调用这个 SQL 命令。）定界符文件里的数据都是用 ASCII 表示的裸数据，没有 SQL 语句，没有注释，也没有列名称。以下就是导出 CSV 格式（Comma-separated values，这种格式是用于表列语言的很好的混合语言）文件的例子：

```

mysql> SELECT * INTO OUTFILE '/tmp/t1.txt'
-> FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
-> LINES TERMINATED BY '\n'
-> FROM test.t1;

```

这个结果文件比 SQL 导出文件更加紧凑，更易于用命令行工具生成，但是它最大的优点还是备份和还原的速度。你仍然可以使用 LOAD DATA INFILE 命令将数据加载到表里去，就像导出时一样：

```
mysql> LOAD DATA INFILE '/tmp/t1.txt'
-> INTO TABLE test.t1
-> FIELDS TERMINATED BY ',' OPTIONALLY ENCLOSED BY '"'
-> LINES TERMINATED BY '\n';
```

这里我们要做一个非正式的测试，用来验证 SQL 文件和界定符文件在备份和还原时各自的速度表现。我们准备了一些合适的数用于此次测试。用来导出的表定义如下：

```
CREATE TABLE load_test (
  col1 date NOT NULL,
  col2 int NOT NULL,
  col3 smallint unsigned NOT NULL,
  col4 mediumint NOT NULL,
  col5 mediumint NOT NULL,
  col6 mediumint NOT NULL,
  col7 decimal(3,1) default NULL,
  col8 varchar(10) NOT NULL default '',
  col9 int NOT NULL,
  PRIMARY KEY (col1,col2)
) ENGINE=InnoDB;
```

这个表里有 1 500 万行数据，在磁盘大致上占用 700MB 空间。表 11-1 比较了两种方式在备份和还原时的性能，你会发现在还原时两者的速度差异非常巨大。

表 11-1：SQL 文件和界定符文件在备份、还原时的耗费时间

| Method | Dump size | Dump time | Restore time |
|----------------|-----------|-----------|--------------|
| SQL dump | 727 MB | 102 sec | 600 sec |
| Delimited dump | 669 MB | 86 sec | 301 sec |

SELECT INTO OUTFILE 方法也存在一些局限性：

- 你只能在 MySQL 运行的机器上把数据都备份到一个文件里。（你可以编写一个程序，让它读取一次 SELECT 的结果，然后写到磁盘里，这个滚动 SELECT INTO OUTFILE 的方法，我们看到有些人正在这么使用。）
- MySQL 必须有权限访问要生成文件的那个目录，因为 MySQL 服务器——不是用户运行的 SQL 命令——它要把数据写入那个文件。
- 出于安全方面的原因，你不能覆盖已存在的文件，不管这个文件的权限是怎么分配的。
- 你无法直接导出一个压缩文件。

并行导出与还原

在多 CPU 的系统上，用并行方式执行备份和还原会更快。这里“并行”的意思是对多个表同时做导出或还原，不是说多个程序在同一个表上同时做各自的操作。若两个程序同时将数据加载到一个表单里，这效果一般来说都不会好。

你无需使用很特异的工具来做并行的备份/还原，只需要手动运行备份工具的多个实例就可以了。市面上确实有一些工具和脚本是特地为这个用途而设计的，例如 Maatkit 的 mk-parallel-dump 和 mysqldump (<http://www.fr3nd.net/projects/mysqldump/>)。在编写本内容的时候，这些工具相对来说还比较新。基准测试显示 mk-parallel-dump 比单纯使用 mysqldump 备份要快好几倍。

在 MySQL 5.1 里,mysqlimport 支持多线程同时导入。你也能在早期的 MySQL 上使用 5.1 版本里的 mysqlimport。如果使用的并行度过高,并行导出与还原实际上可能会耗费更长的时间,而且还会引起更多的数据碎片,从而影响到系统的性能。

11.4.2 文件系统快照

Filesystem Snapshots

文件系统快照是做在线备份的极佳方法。有快照功能的文件系统能够在瞬间将它的内容做一个一致的镜像,这个镜像你可以用来做备份。有快照功能的文件系统和设备包括 FreeBSD 的文件系统、ZFS 文件系统、GNU/Linux 的逻辑管理器 (Logical Volume Manager, LVM), 以及许多 SAN 系统和文件存储解决方案 (例如 NetApp 存储设备)。

在这里,不要把快照跟备份弄混了。做一个快照只是减少锁定时间的一个简单手段,在释放锁之后,你必须把这些文件都复制到备份里去。事实上,你可以在 InnoDB 上选择做无需锁定的快照。在你所希望的最少锁定甚至不用锁定的前提下,我们为你展示两种使用 LVM 在全 InnoDB 数据库上做备份的方法:



提示: Lenz Grimmer 的 mylvmbakup 是一个现成的 Perl 脚本,它通过 LVM 来创建 MySQL 备份。更多内容请查看本章后面第 511 页的“备份工具”。

LVM 快照是怎么工作的

LVM 使用了写时复制 (Copy-on-write) 技术来创建快照——也就是即时获得全部卷的一个逻辑副本。这有点像数据库里的 MVCC,除了它只保留一份旧版本的数据外。

你应该已经注意到我们没说是一个物理副本。一个逻辑副本看上去跟已做了快照的卷有一模一样的数据,但是在开始之初,它并不包含任何数据。LVM 只是简单地标注了一下你做快照的时间,而不是把数据都复制到快照里。当你需要从快照里读取数据时,它才会到数据来源的卷里把它们都读出来。所以,最初的副本本质上是一个即时的操作,无论你要做快照的卷有多大。

当卷里的数据在快照完成之后发生变动时,LVM 就会在变动生效之前,把受影响的数据块复制到一个为快照保留的存储区域里,LVM 不会保留多个“旧版本”的数据,因此,同一个数据块上的多次变动不会让快照再去复制数据了。换句话说,每一个数据库只有在第一次写入时才会触发写时复制,从而把数据复制到保留区域里。

现在,当你要读取快照里这些已被复制过的数据块时,LVM 就会从保留存储区域里读取数据,而不是最初来源的卷里。这样可以使你能在快照里继续读取那份数据,而不会妨碍来源卷里的任何操作。图 11-1 描述了这个安排方式。

快照会在 /dev 目录下新建一个逻辑设备,你可以把这个设备挂载 (Mount) 到任何你要挂载的地方。

通过这项技术,在理论上你也可以对非常巨大的卷做快照,而耗费的物理存储空间依然很小。但是,你要预留出足够的空间来容纳当来源卷里的数据更改时要复制过来的数据块。如果保留的写时复制空间不够用,快照会耗尽空间,从而变成不可用,就像拔掉了一个外部驱动器:从中读数据的备份任务都会因 I/O 错误而停止。

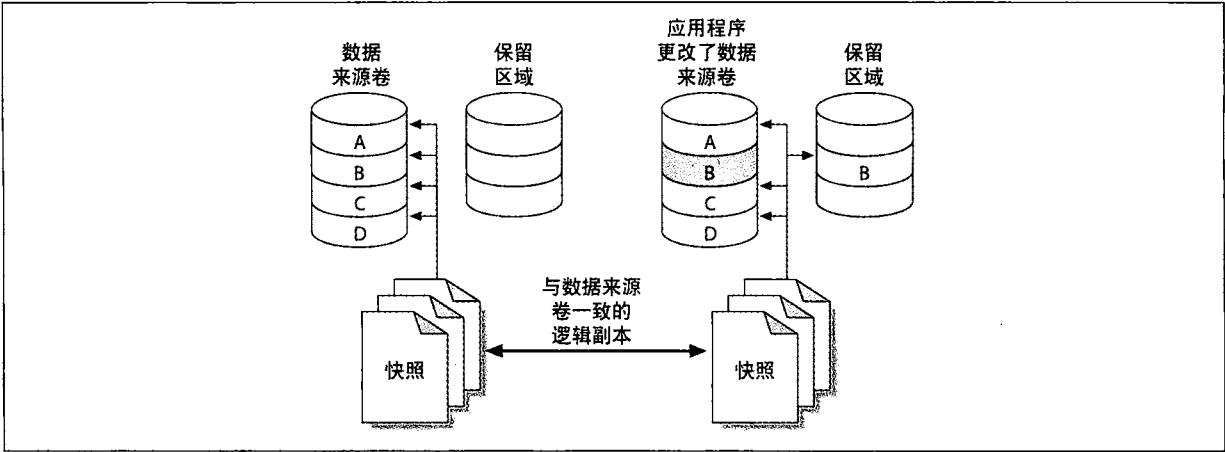


图 11-1：写时复制技术是如何减少快照所需的大小的

先决条件和配置方法

在你掌握磁盘上的分卷信息前来谈创建一个快照几乎是没有什么意义的。你需要先确定你的系统是否已经做好了配置，有了这些配置你才可以在某个即时点上做备份时，得到一个包含所有文件的一致性副本。所以，首先要确认你的系统是否已经符合了以下条件：

- 所有 InnoDB 文件（InnoDB 表空间文件和 InnoDB 事务日志）必须在一个逻辑卷（分区）里。需要即时点上的数据具有一致性，LVM 无法在同一个时间内为一个以上的卷做符合一致性的快照。（这是 LVM 的局限性，另外一些系统没这样的问题。）
- 如果你也需要备份表定义，那么，MySQL 数据目录必须在同一个逻辑卷里。如果你使用另外一个方法来备份表定义，例如把只含有样式的备份记录放入版本控制系统里，就无须担心这样的问题。
- 你在分卷组里必须有足够的空闲空间来创建快照。空间的大小依赖于工作负荷。当你搭建系统时，留一些未分配的空间，这些可以在后来被用作快照存储。

494

LVM 有一个分卷组（Volume group）的概念，它包含一个或多个逻辑分卷。你可以在你的系统里看到那些分卷组，像下面这样：

```
# vgs
VG      #PV #LV #SN Attr   VSize  VFree
vg      1   4   0 wz--n- 534.18G 249.18G
```

这个输出结果显示一个分卷组有 4 个逻辑卷分布在一个物理卷上，还留有 250GB 的空闲空间。如果你有需要，vgdisplay 命令能给你更多的细节。现在，就让我们看一下系统里的逻辑卷：

```
# lvs
LV      VG      Attr   LSize   Origin Snap%  Move Log Copy%
home    vg      -wi-ao 40.00G
mysql   vg      -wi-ao 225.00G
tmp     vg      -wi-ao 10.00G
var     vg      -wi-ao 10.00G
```

这个输出结果显示 mysql 卷上有 225GB 空间。设备是/dev/vg/mysql，这只是一个名称，尽管看上去像个文件系统里的路径。更使人混淆的是，从这个名称的设备到真实的设备节点上还有一个叫/dev/mapper/vg-mysql 的

符号化的链接。你可以使用 `ls` 和 `mount` 命令看到这个链接：

```
# ls -l /dev/vg/mysql
lrwxrwxrwx 1 root root 20 Sep 19 13:08 /dev/vg/mysql -> /dev/mapper/vg-mysql
# mount | grep mysql
/dev/mapper/vg-mysql on /var/lib/mysql type reiserfs (rw,noatime,notail)
```

掌握了这些信息以后，你就可以创建文件系统快照了。

创建、挂接和移除 LVM 快照

使用一个单独的命令，你就可以创建快照了。你只需要决定哪里存放快照、给写时复制分配多少存储空间。在分配时要毫不犹豫地比预想的再多分配一些空间。LVM 使用存储空间时不像你指定的那样，它会为未来的使用事先保留起一部分，因此，多保留一些空间总是无害的，除非你需要把这些空间用于同时做的另一些快照。

这里我们就练习一下创建快照。我们给写时复制分配 16GB 的空间，并命名为 `backup_mysql`：

```
# lvcreate --size 16G --snapshot --name backup_mysql /dev/vg/mysql
Logical volume "backup_mysql" created
```



提示：我们把卷名叫做 `backup_mysql`，而不是 `mysql_backup`，这样就可以避免 `tab` 自动完成时的二义性，从而也避免了不小心删除 `mysql` 分组卷的可能。类似这样的细节可以帮你避免大灾难。至少，本书的一个作者就曾因为一个草率的 `tab` 自动完成，把 LVM 快照毁掉了。

现在让我们看一下最近创建的分卷状态：

```
# lvs
LV          VG      Attr   LSize   Origin Snap%   Move Log Copy%
backup_mysql vg      swi-a- 16.00G   mysql    0.01
home        vg      -wi-ao 40.00G
mysql       vg      owi-ao 225.00G
tmp         vg      -wi-ao 10.00G
var         vg      -wi-ao 10.00G
```

我们注意到快照的属性已经跟当初的设备不一样了，这些输出信息里还包含了一些额外的信息：那块用于写时复制的 16GB 空间开始时的使用率和当前使用率。在备份的时候监控这些信息的变化是个很好的主意。这样你就能看到空间是否快要满了、是否快要宕机了。你也可以通过监控系统，例如 Nagios 来监控设备的状态：

```
# watch 'lvs | grep backup'
```

就像你刚才在 `mount` 输出信息里看到的那样，`mysql` 卷里包含着一个 ReiserFS 文件系统。这意味着快照卷也是这个文件系统，你可以像其他文件系统那样挂接并使用它：

```
# mkdir /tmp/backup
# mount /dev/mapper/vg-backup_mysql /tmp/backup
# ls -l /tmp/backup/mysql
total 5336
-rw-r----- 1 mysql mysql    0 Nov 17 2006 columns_priv.MYD
-rw-r----- 1 mysql mysql 1024 Mar 24 2007 columns_priv.MYI
-rw-r----- 1 mysql mysql 8820 Mar 24 2007 columns_priv.frm
-rw-r----- 1 mysql mysql 10512 Jul 12 10:26 db.MYD
-rw-r----- 1 mysql mysql 4096 Jul 12 10:29 db.MYI
-rw-r----- 1 mysql mysql 9494 Mar 24 2007 db.frm
... (下略) ...
```

这仅仅是一个练习，所以，最后我们使用 `lvremove` 命令把快照从文件系统上卸下来并删除。

```
# umount /tmp/backup
# rmdir /tmp/backup
# lvremove --force /dev/vg/backup_mysql
Logical volume "backup_mysql" successfully removed
```

用于在线备份的 LVM 快照

你已经看到了怎样去创建、挂接和删除快照，现在你能用它们来做备份了。首先，让我们看一下如何在不停止 MySQL 服务器的情况下备份一个 InnoDB 数据库，接着连接到 MySQL 服务器，通过一个全局的读锁把表的缓冲区都刷新到磁盘上，然后得到一个二进制日志的备份记录：

```
mysql> FLUSH TABLES WITH READ LOCK; SHOW MASTER STATUS;
```

记录 `SHOW MASTER STATUS` 的输出信息，确认你还连接着 MySQL 服务器没有释放掉锁。接下来就是做 LVM 快照，做完后立即使用 `UNLOCK TABLES` 或关闭数据库连接把读锁释放掉。最后，把快照挂接上去，复制里面的文件到备份位置上。如果你是用脚本来做的，那么锁定的时间能减少到几秒钟。

这个方法的主要问题是获取读锁可能会花一些时间，特别是当一个很长的查询正在执行时。当数据库连接在等待一个全局读锁时，所有的查询都会被堵塞，也不可能预计这个过程会有多久。

用 LVM 快照做 InnoDB 的无锁备份

无锁备份跟上面的方法相比有点不同。这个差别在于你用不着做一次 `FLUSH TABLES WITH READ LOCK`，这也意味着在你磁盘上的 MyISAM 文件无法保证具有一致性，但是对于 InnoDB 来说，这可能不是问题。你仍然会有一些 MyISAM 表在 `mysql` 的系统数据库里，但是，如果你的备份工作比较典型，这些表在你做快照的时候，可能也不会发生改变。

如果认为那些 `mysql` 系统表也可能会改变，你可以先锁定它们，然后刷新它们的缓冲区。之后，就不要在这些表上做耗时很长的查询了，这样备份起来会快很多：

```
mysql> LOCK TABLES mysql.user READ, mysql.db READ, ...;
mysql> FLUSH TABLES mysql.user, mysql.db, ...;
```

如果你没有得到一个全局读锁，使用 `SHOW MASTER STATUS` 也不能得到什么有用的信息。然而，当在快照上启动 MySQL 时（为了检验备份的完整性），你会在日志文件里看到类似下面这样的信息：

```
InnoDB: Doing recovery: scanned up to log sequence number 0 40817239
InnoDB: Starting an apply batch of log records to the database...
InnoDB: Progress in percents: 3 4 5 6 ...[omitted]... 97 98 99
InnoDB: Apply batch completed
InnoDB: Last MySQL binlog file position 0 3304937, file name /var/log/mysql/mysql-
bin.000001
070928 14:08:42 InnoDB: Started; log sequence number 0 40817239
```

文件系统快照与 InnoDB

即使在你锁定了所有的表之后，InnoDB 的后台线程还在继续工作，也就是说，当你正在做快照的时候，文件可能还正在被写入。因为 InnoDB 没有执行它的关闭程序，所以，快照里的 InnoDB 文件看上去就像所在的服务器遭遇了意外断电。

但这不算个问题，因为 InnoDB 是一个 ACID 系统，在任何时刻（例如你做快照的那一刻），每个提交了的事务要么在 InnoDB 的数据文件里，要么在日志文件里。当你恢复了快照，重新启动 MySQL 之后，InnoDB 会运行一个还原过程，就像服务器意外断电之后那样。它会在事务日志里查找到还未更新到数据文件的事务，然后将它们写到数据文件里，这样就可以保证不会丢失任何事务。这就是为什么必须强制在 InnoDB 里的数据和日志文件一起做快照。

这也是为什么在备份之后你要测试备份效果：启动一个 MySQL 的实例，加入那个新做的备份，运行 InnoDB 的还原功能，让它核对所有的表。使用这个方法，你就不会备份损坏的数据，哪怕你事先根本不知道。（文件会有种种原因而损坏。）另外一个好处是将来从备份记录恢复数据的过程会更快，因为你已经运行过还原过程了。

在将快照复制到备份里之前，你可以选择运行这个过程，但是，这也会给系统增加一点开销，这取决于你是否事先做了计划。（更多的内容会在后面展开。）

InnoDB 记录了 MySQL 二进制日志的位置，该位置对应于它被恢复的那个点。这个日志位置可以被用来做即时还原。

这种使用快照的无锁备份方法在 MySQL 5.0 及更早的版本里有一点扭曲。这些版本的 MySQL 使用 XA 来协调 InnoDB 和二进制日志之间的事务。如果你把备份恢复到不同 `server_id` 的服务器上（这个 `server_id` 是属于备份时的那台服务器的），这台服务器会发现已准备好的事务的 ID 跟它自己的不匹配。对于这种情况，服务器会感到迷惑，当做还原时，那些事务可能会停滞在 `PREPARED` 状态上。这虽然很少发生，但一直存在着这种可能。这就是为什么在你认为备份成功前，最好检验一下备份记录：它可能没法被还原！

如果你正在一个从服务器上做快照，InnoDB 还原会打印出几行信息，类似于下面这个样子：

```
InnoDB: In a MySQL replication slave the last master binlog file
InnoDB: position 0 115, file name mysql-bin.001717
```

在一些版本的 MySQL 里，这几行输出信息告诉你主服务器的二进制日志的定位（跟从服务器上的日志定位），在该点上 InnoDB 做了还原，这个对于从服务器上做备份或从服务器之间的克隆很有用处。然而，在 MySQL 5.0 及更早的版本里，这些数值是靠不住的。

498 规划 LVM 备份

LVM 快照备份对于系统性能并非没有影响，服务器往来源卷里写入的数据越多，备份它们的开销就越大。当服务器以随机次序更新几个完全不同的数据块时，磁盘的磁头必须来回寻找写时复制的空间，并把旧数据写入这个区域。从快照里读数据也需要开销，因为对于大多数数据来说，LVM 还是要真正地到来源卷里去读取。从写时复制里读取只是在需要的时候才会发生；因此，一个从快照里按逻辑顺序读数据的过程，实际上会引起磁盘磁头的来回移动。

你要为此类事情的发生提早做规划。要考虑的事情还有：如果你使用了大量的写时复制空间，那么来源卷和快照的性能会比通常的读写还要糟糕。这不仅能减缓 MySQL 服务器的运行速度，而且还会减缓备份时的文件复制进程。

另一个重要的事情是要为快照分配足够多的空间。我们使用的是下面这个方法：

- 要记住 LVM 只要把更新的数据块复制到快照里一次就行了。当 MySQL 在来源卷里写入数据块时，它会把原来的数据都复制到快照里去，然后在它的例外表里标注上这个被复制的数据块。将来对这个数据块的任何写入操作都不会触发“复制到快照”这个操作了。
- 如果你只是用 InnoDB，要考虑到 InnoDB 是怎么写数据的。因为它把所有数据都写两遍，至少有一半的 InnoDB 写 I/O 是写到了双写缓冲区（Doublewrite Buffer）、日志文件和磁盘上的相关的小区域里。这些写入都会一次次地重用磁盘上的同一个数据块空间。因此，它们对快照最初建立的时候会有影响，但是之后，就不会再因快照引发写操作了。
- 下一步是估计在数据块被复制到快照之前，你在数据块上写入时 I/O 的数量会是多少（这里不是指对同一份数据所作的一次又一次的更改）。估算要宽松一点。估算出来的值就是你预计快照要产生额外 I/O 的数量。（还要加上一点点 LVM 进程所要使用的数量。）
- 使用 vmstat 或 iostat 来获取统计信息，以获知服务器在每一秒里在写多少个数据块。有关这方面的工具，请查看第 7 章。
- 测量（或估算）你把备份复制到别的位置上所花费的时间，换句话说，你的 LVM 快照为了被复制要打开多久。

让我们假设你已经估算出写操作的一半会引发快照的写时复制，服务器每秒钟写 10MB 数据，如果要把备份复制到另外一台服务器上，它要花费 1 小时（3 600 秒）的时间。这样，你将需要 $1/2 \times 10\text{MB} \times 3600$ ，即 18GB 的空间用于快照。谨慎地考虑到警告之类的信息，还可以再增加一些空间。

有时很容易算出打开快照时多少数据会被改掉。让我们再回到在别处多次用到过的一个例子，BoardReader 论坛搜索引擎在每个节点上大约有 1TB 的 InnoDB 表，然而，我们知道这里最大的开销是载入新数据。每天大约有 10GB 的新数据会被加进来，因此，对于快照而言，50GB 才算是充分的空间。这个估算也不总是准确的，在一个点上，我们会运行一个耗时很长的 ALTER TABLE 的命令，一个接一个地修改数据分块，每一个分块上的更改还会涉及超过 50GB 的数据。当这个命令运行时，我们就无法做备份了。

其他用途和替代方法

你可以把快照不仅仅用作备份。举例来说，它们可作为一个有潜在危险操作之前的“检查点”。有一些系统，比如 ZFS，能让你将快照升级为原始的文件系统。这有助于你把当前的数据回滚到做过快照的时间点上。

文件系统快照也不是取得数据即时副本的唯一途径。另外还有一个可选择的方法是 RAID 分割：如果有一个 3 碟软件 RAID 镜像，举例来说，你可以将其中一碟从镜像上移下来，然后单独把它挂接上。这里就没有了写时复制时系统资源的“罚款”，如有必要，还能很方便地把这种“快照”升级为主文件。

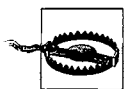
11.5 从备份中还原

还原是真正重要的事情。在这一节里，我们将集中于 MySQL 类型的还原，并且假定你知道如何处理环境里的其他问题。要例行公事地进行“消防演习”，这样在真正的紧急状况发生时，你就知道如何去备份和还原数据。首先要做的是测试备份记录。

怎样还原你的数据取决于你如何备份数据。你可能需要完成下面的全部或部分步骤：

- 关闭 MySQL 服务器。
- 在服务器配置和文件权限上做好记录。
- 把数据从备份记录移到 MySQL 数据目录下。
- 更改配置。
- 更改文件权限。
- 在有限访问权下重新启动服务器，然后等它完全启动。
- 重新加载逻辑备份文件。
- 检查和重放二进制日志。
- 核实恢复出来的内容。
- 使用完全访问来重启服务器。

505 在接下来的各小节里，我们会演示如何按需要来操作每一个步骤。对于某些备份方法或工具，我们会增加注释，表明这些方法或工具在本章的后面部分里会出现。



警告：如果你可能还会需要当前版本的文件，那就不要让还原出来的文件覆盖它们。举例来说，如果备份记录里包含了二进制日志，你需要重放日志来做即时点还原，那就不要让当前最新的日志被备份记录里的旧版本日志覆盖了。如果必要，就把他们重命名或者移到别的位置上去。

11.5.1 限制访问 MySQL

在还原时，除了还原进程之外不让其他一切程序来访问 MySQL，有时也很重要。在复杂的系统里，这一点就很难被保证。我们喜欢在启动 MySQL 时使用 `--skip-networking` 和 `--socket=tmp/mysql_recover.sock` 选项，这可以保证任何现存的应用不会来访问 MySQL，直到我们对 MySQL 检查完毕并上线。这对于逻辑备份尤其重要，因为它是分成一块块被加载的。

11.5.2 还原裸文件

还原裸文件相当的直接——也就是说它没有太多选项可以用。这是好事情也是坏事情，具体要根据还原的要求而定，通常是把文件复制到指定位置就可以了。

是否需要关闭 MySQL 取决于使用的存储引擎。MyISAM 的文件一般都是相互独立的，只要复制每个表的 .frm、.MYI 和 .MYD 文件就可以了，哪怕服务器仍在运行。一旦有人要查询这些表，或者服务器要查找表时（比如执行 `SHOW TABLES` 命令），服务器就会自动发现这些新加进来的表。如果在复制表文件的时候，那个表正打开着，那可能会引起点麻烦，因此，在复制表文件之前，你应该把这些表删除掉，或者重命名，或者干脆使用 `LOCK TABLES` 和 `FLUSH TABLES` 把它们关闭。

InnoDB 是另外一个麻烦。如果你正在还原一个通常的 InnoDB 数据库，其中的表全都存放在一个单独的表空间里。这样，你必须先关闭 MySQL，把这些文件复制或移动到另外的地方，然后重启 MySQL。你也需要确认 InnoDB

的事务日志文件与表空间文件是相匹配的。如果两者不匹配——例如你替换了表空间文件，但是没动事务日志文件——InnoDB 会拒绝启动。这就是数据文件跟日志文件一起备份的紧要性之一。

如果你正在使用的是新版 InnoDB 一表一文件结构 (`innodb_file_per_table`)，InnoDB 会把数据和索引存放在一个 .ibd 文件里，这个文件类似于 MyISAM 里的 .MYI 和 .MYD 文件的合体。现在，你也可以在服务器还在运行时，通过复制文件来对单独的表做备份和还原操作了。但是，它还是没有 MyISAM 那么简单，那个单独文件不能完全独立于 InnoDB 而存在。每一个 .ibd 文件包含了一些内部信息，它们告诉 InnoDB 如何将它们与主（共享）表空间联系起来。当你还原这样一个文件时，必须要告诉 InnoDB 去“导入”这个文件。

在这个还原过程里有很多约束条件，具体的可以阅读 MySQL 帮助手册里的关于使用“一表一个表空间”的章节。这些约束条件里，最大的约束是你只能将表还原到备份出来的服务器上。用这样的配置条件来备份和还原表也不是不可能，只是要比你想象的更麻烦一些。

所有这些复杂性意味着还原裸文件是枯燥乏味的，而且还容易出错。一个很好的经验法则是：还原过程越艰难复杂，你就越要保护好逻辑备份。在裸文件备份出了点问题，不能确定是否可以供 MySQL 使用的情况下，事先保存一份逻辑备份是个不错的主意。

在还原裸文件后启动 MySQL

在做完还原之后，启动 MySQL 服务器之前，你还需要做一些事情。

首要的，也是最容易被遗忘的问题之一，就是检查服务器配置，在启动 MySQL 服务器之前，确认那些还原的文件有正确的属主（Owner）及权限。这些属性必须正确，否则 MySQL 就没法启动了。这些属性在服务器之间各不相同，因此要翻一下笔记，看看是否都是你要设置的属性。典型地，你要让 mysql 的用户和组拥有这些文件和目录，只有他们才能读和写，不能有其他入。

我们也建议在服务器启动时，查看 MySQL 错误日志。在一个 Unix 类型的系统里，你可以这样来查看日志：

```
$ tail -f /var/log/mysql/mysql.err
```

错误日志的确切位置在各系统里有所不同。一旦能够监控到日志，你就可以开始启动 MySQL，然后查看是否有错误发生。如果一切进展顺利，那你就拥有了一台被完美还原的服务器。

在新版的 MySQL 里，查看错误日志更显得重要。老版本的 MySQL 在 InnoDB 有错误时，根本不会启动，但是在新版里，服务器仍然会启动，它只是把 InnoDB 关闭了。即使服务器看起来没有任何问题地启动了，你还是应该在每个数据库里运行 `SHOW TABLE STATUS` 查看一下，然后再检查一遍错误日志。

11.5.3 还原逻辑备份

如果你使用逻辑备份来还原，而不是裸文件，那你就要用 MySQL 服务器本身把数据加载到表里去，不再是简单地用操作系统把文件复制到某个地方。

在加载那个导出的文件之前，你需要花些时间考虑一下它有多大、需要多久才能加载上去，以及其他任何需要在加载前考虑的问题（例如通知用户、关闭应用的一部分功能）。临时关闭二进制日志也是个好主意，除非你要把还原过程复制到从服务器上去：一个巨大的导出文件很难被服务器加载，更何况把它写到日志里，这会增加更多（很可能是不必要）的系统开销。对于某些存储引擎来说，加载大文件也有个顺序问题。举例来说，在一

个单独的事务里把 100GB 的数据加载到 InnoDB 里绝不是个好主意，因为这会导致一个绝大的回滚段。你应该把文件分成一块块容易管理的备份数据，然后每一块一个事务，最终把文件全部加载上去。

你可能会用到两种还原方法以对应于你做的逻辑备份的类型。

加载 SQL 文件

如果你有一个 SQL 导出文件，文件里包含了可执行的 SQL 语句。你要做的就是运行它们。假定你把 Sakila 示例数据库和式样备份到一个单独的文件里，下面这行典型的命令就是你用来还原的：

```
$ mysql < sakila-backup.sql
```

你也可以在 mysql 命令行客户端上用 SOURCE 命令来加载 SQL 文件。对于同一个事情，虽然这显得很不一样，但是它让一些事情变得简单了。举例来说，如果你是 MySQL 上的管理员用户，你可以直接关闭日志（原先还需要通过客户端连接来关闭），然后就可以加载文件而无需重新启动 MySQL 服务器：

```
mysql> SET SQL_LOG_BIN = 0;
mysql> SOURCE sakila-backup.sql;
mysql> SET SQL_LOG_BIN = 1;
```

如果你使用 SOURCE，那就要注意在执行过程中如果发生一个错误，不会使整批语句退出执行，默认情况是只有当你把文件重定向到 mysql 后才会退出。

如果压缩了备份记录，不要分开来做解压缩和加载。它们可以用一个单独的操作来完成，这样会快很多：

```
$ gunzip -c sakila-backup.sql.gz | mysql
```

如果你用 SOURCE 命令加载一个压缩了的文件，请阅读下一节里关于命名管道的讨论。

若只还原一张单独的表（例如 actor 表）又会怎样？如果数据没有换行符，而且样式已经固定下来，那也不难还原：

```
$ grep 'INSERT INTO `actor`' sakila-backup.sql | mysql sakila
```

或者，文件是压缩的：

```
$ gunzip -c sakila-backup.sql.gz | grep 'INSERT INTO `actor`' | mysql sakila
```

如果还原数据时要创建表，而那个单独文件又包含了全部的数据库，这样，你就不得不编辑这个文件了。这就是为什么很多人喜欢把每个表导出到各自对应的文件里。许多编辑器无法处理这样的巨大文件，特别是当它们被压缩过的时候。更何况，你不是真地想要编辑那个文件——你只想取出相关的几行——这样就可以用命令行来做余下的工作了。使用 grep 能很容易地从指定的表里取出 INSERT 语句，就像我们在前一个命令里做的那样。但是，取得 CREATE TABLE 语句会更难一些。这里有一个 sed 脚本可以帮你取出想要的段落：

```
$ sed -e '/.{H;$!d;}' -e 'x;/CREATE TABLE `actor`/!d;q' sakila-backup.sql
```

我们承认这个命令的语义看上去很模糊。如果你非得用这种方式来还原数据，那备份记录一定是设计糟糕的。稍微有一点计划性，就可能防止你陷入这样的境地：你恐慌万分，拼命想找出 sed 是怎样工作的。其实只要事先将每个表备份到它们自己的文件里就行了，或者，更好一点，把数据和样式分开来备份。

加载定界符文件

如果你是通过 SELECT INTO OUTFILE 来导出数据的，就必须用带同样参数的 LOAD DATA INFILE 来做还原。你也可以使用 mysqlimport——它是 LOAD DATA INFILE 的一个封装，依赖于命名惯例来决定从哪里加载文件的数据。

我们希望你导出数据样式，而不仅仅是数据。如果这样做了，它就是一个 SQL 导出文件了，你可以用上一节讲到的技术来还原它。

在 LOAD DATA INFILE 里有一个巨大的你可以用到的优化。它必须从文件里直接读取，因此，你可能会认为必须在加载前先做解压缩。这其实是个非常缓慢，又是磁盘密集的操作过程。但是这里还有一个变通的办法——至少在支持 FIFO “命名管道”文件的系统上（例如 GNU/Linux）是可行的。首先创建一个命名管道和流，并把数据压缩到里面：

```
$ mkfifo /tmp/backup/default/sakila/payment.fifo
$ chmod 666 /tmp/backup/default/sakila/payment.fifo
$ gunzip -c /tmp/backup/default/sakila/payment.txt.gz > /tmp/backup/default/sakila/
payment.fifo
```

要注意到：在这里，我们使用了一个大于号 (>) 把解压缩结果重定向到了 payment.fifo 文件——这不是一个管道记号（如果是这种记号，就会在两个程序之间创建匿名管道）。payment.fifo 本身就是一个命名管道，所以这里不需要使用匿名管道。

这个管道会一直等待，直到某个程序在另一端把它打开，读取数据。这是很清晰的一个过程：MySQL 能够从管道里读取压缩了的数据，就像从其他文件里读取一样。如果考虑周全，就不要忘记关闭二进制日志：

504

```
mysql> SET SQL_LOG_BIN = 0; -- Optional
-> LOAD DATA INFILE '/tmp/backup/default/sakila/payment.fifo'
-> INTO TABLE sakila.payment;
Query OK, 16049 rows affected (2.29 sec)
Records: 16049 Deleted: 0 Skipped: 0 Warnings: 0
```

一旦 MySQL 加载完数据，gunzip 就退出了，然后，你可以删除那个命名管道。你可以使用同样的技术在 MySQL 命令行客户端上用 SOURCE 命令来加载压缩文件。

为什么要测试备份？

本书的作者之一最近为了节省空间，让数据处理能更快一些，把表里某列的数据类型从 DATETIME 改为 TIMESTAMP——就像在第 3 章里推荐的那样。这种表最终的定义方式如下：

```
CREATE TABLE tbl1 (
  col1 timestamp NOT NULL,
  col2 timestamp NOT NULL default CURRENT_TIMESTAMP
    on update CURRENT_TIMESTAMP,
  ... more columns ...
);
```

此表的定义在 MySQL 5.0.40（就是创建这个表时所在的服务器）里会引起一个语法错误。于是，你可以导出它，但是无法重新加载它。像这样怪异、无法预料的错误就是要测试备份记录的很重要的原因之一。你永远无法知道有什么来妨碍你还原数据。

11.5.4 即时点还原

Point-In-Time Recovery



MySQL 上最常用的即时点还原是还原最近的一个完全备份记录，然后重放这个备份记录以来的二进制日志（有时把这个做法叫做“前滚还原”）。有了二进制日志，你就可以在任何想要的点上来做还原。你甚至可以在没太多麻烦的情况下还原整个数据库。

一个常见的场景是在一个有害的操作后，要消除它所带来的影响，例如 DROP TABLE 操作。让我们举一个简化了的例子来说明只用 MyISAM 表是怎么做到这一点的。假定事情发生在午夜，备份进程正在运行一个命令（相当于下面这样一些语句），它的用途就是把数据库复制到同一服务器的另外一个位置上：

```
mysql> FLUSH TABLES WITH READ LOCK;
-> server1# cp -a /var/lib/mysql/sakila /backup/sakila;
mysql> FLUSH LOGS;
-> server1# mysql -e "SHOW MASTER STATUS" --vertical > /backup/master.info;
mysql> UNLOCK TABLES;
```

第二天，假设某人运行了下面这样的语句：

```
mysql> USE sakila;
mysql> DROP TABLE sakila.payment;
```

为了便于说明，我们假设能在数据库隔离的状态下还原它（这也就是说该数据库里没有表涉及跨数据库查询）。我们还假设要在一段时间后才发现有人执行了这段不良的语句。这个例子的目标是还原受这段语句影响的所有数据，我们必须保留其他表的更新操作，包括在执行了这段语句后的那些操作。

这还不是所有难以处理的部分。首先，我们要停掉 MySQL 以防止有新的数据操作发生，然后只从备份里还原出 Sakila 数据库：

```
server1# /etc/init.d/mysql stop
server1# mv /var/lib/mysql/sakila /var/lib/mysql/sakila.tmp
server1# cp -a /backup/sakila /var/lib/mysql
```

我们在 my.cnf 里加入下面这两行参数来关闭那些普通的数据库连接：

```
skip-networking
socket=/tmp/mysql_recover.sock
```

现在就可以安全地启动服务器了：

```
server1# /etc/init.d/mysql start
```

接下来的任务就是在日志中找到我们要重放和要略过的语句。事实上，既然备份是在午夜时分生成的，服务器就只创建了一个日志。我们可以用 grep 来检查这个日志文件，找出那段不良的语句来：

```
server1# mysqlbinlog --database=sakila /var/log/mysql/mysql-bin.000215 | grep -B
3 -i 'drop table sakila.payment'
# at 352
#070919 16:11:23 server id 1 end_log_pos 429 Query thread_id=16 exec_time=0
error_code=0
SET TIMESTAMP=1190232683/*!*/;
DROP TABLE sakila.payment/*!*/;
```

我们要略过的语句在日志的 352 位置上，紧接着的语句是在 429 位置上。于是，我们可以把日志从开始重放到 352 位置，然后再从 429 开始到最后。具体命令如下：

```
server1# mysqlbinlog --database=sakila /var/log/mysql/mysql-bin.000215
--stop-position=352 | mysql -uroot -p
server1# mysqlbinlog --database=sakila /var/log/mysql/mysql-bin.000215
--start-position=429 | mysql -uroot -p
```

现在我们要做的就是检查一下以确保结果正确，然后关闭服务器，把 my.cnf 的配置改回来，再重新启动服务器。

11.5.5 更高级的还原技术

复制和即时点还原使用的是同一个机制：服务器的二进制日志。这意味着从一些不太明显的角度来看，复制在还原过程中也可以是个很有帮助的工具。在这一节里，我们会展示其中的一些可能性。这不是一个面面俱到的目录，但是它可以在你设计还原过程时提供一些参考。你要记得把在还原过程中要做的事情，先写下来演练一遍。

把延时复制用于快速还原

如同我们在本章前面所提到的那样，如果你能在从服务器执行那些可恶的语句之前发现这个问题，使用延迟复制从服务器能够使即时点还原更快、更容易。

这个过程跟前面小节里讲的有一点点不同，但是，理念还是一样的。你要先停掉从服务器，然后使用 `START SLAVE UNTIL` 来重放事件，直到那个要略过的语句之前。接着，你执行 `SET GLOBAL SQL_SLAVE_SKIP_COUNTER=1` 来跳过这个坏语句。如果你要跳过好几个语句，就把 1 改成更大的数字（或者可以简单地使用 `CHANGE MASTER TO` 在日志里移位）。

接下来你要做的就是执行 `START SLAVE`，让从服务器运行起来，直到把日志都重放完毕。到此，你的从服务器已经做完了即时点还原里的所有脏活累活。现在，你可以把从服务器提升为主服务器了，刚才的还原也只有很少一点中断服务的时间。

你甚至都不需要一个延迟复制从服务器来加速还原过程，从服务器本身就很有用，因为从主服务器上获取日志，放在了另外一台机器上。如果主服务器的磁盘发生故障，那么从服务器上的重放日志就可能是唯一能信得过的最新的主服务器日志副本了。（把二进制日志放在一个 SAN 里，或者用 DRBD 复制它们，会显得更加安全，这在第 9 章里已经讨论过了。）

用日志服务器来还原

这里是本书作者将复制用于还原的方法：搭建一台日志服务器。（具体做法的细节请查看第 374 页的“构建一台日志服务器”。）

日志服务器有更好的兼容性，用在还原上比 `mysqlbinlog` 更方便。这不仅是因为有 `START SLAVE UNTIL` 选项，还因为你可以设置复制规则（例如 `replicate-do-table`）。使用日志服务器，你能做比原来在做的还要复杂的过滤。

举例来说，日志服务器可以让你很轻松地恢复一个单独的表。这本来对于 `mysqlbinlog` 和命令行工具来说很难做到的——事实上，它确实很难，我们也建议你不要去尝试了。

让我们假设之前有一个粗心的开发人员把一个表删除了，现在我们想把它还原出来，但又不打算把整个服务器恢复到昨天晚上的那个备份状态。下面就是如何用日志服务器来处理这个事情：

1. 把你要还原的那台服务器叫做服务器 1。
2. 把昨晚的备份还原到另外一台服务器上（就叫它服务器 2）。在这台服务器上运行还原过程，以免在还原过程中出错把事情搞得更糟。
3. 按照第 374 页的“搭建日志服务器”指引，搭建一台针对于服务器 1 日志的日志服务器。（为更加小心，有个好主意是：把日志复制到另外的服务器上，然后在那里搭建日志服务器。）
4. 将服务器 2 的配置文件作如下修改：

```
replicate-do-table=sakila.payment
```
5. 重新启动服务器 2，使用 `CHANGE MASTER TO` 命令，将它作为日志服务器的从服务器。将它配置为从昨晚备份记录的日志定位位置上读取数据，这时还不能运行 `START SLAVE`。
6. 检查服务器 2 上 `SHOW SLAVESTATUS` 的输出项，核对数据是否都正确了。在这方面一定要三思而后行。
7. 找到那段不良语句在日志里的位置，然后在服务器 2 上执行 `START SLAVE UNTIL` 来重放事件，直到那个位置之前。
8. 用 `STOP SLAVE` 停止服务器 2 上的进程。现在，那个被删除的表应该已经还原回来了。
9. 把这个表从服务器 2 复制到服务器 1 上。

这个过程只有当服务器不涉及多表 `UPDATE`、`DELETE` 和 `INSERT` 操作时才是可能的。以上 3 种操作的任何一种都会使数据库产生不同的状态，跟原来单独一个表操作时产生的日志不一样。因此，这个表还原出来的数据可能会跟它应有的不尽相同。

11.5.6 InnoDB 还原

InnoDB 在每次启动时都会检查数据和日志文件，以确定是否需要还原。然而，InnoDB 的还原跟我们在本章里讲的还原又是两码事。它不是还原原先备份的数据，而是根据日志里的事务状态，把数据更新到数据文件，或者作回滚。

InnoDB 真正在做的还原工作其实要比描述的更复杂。我们关注的焦点是当 InnoDB 发生严重问题时，该怎么去做还原。

多数时候，InnoDB 都善于自己修复问题。除非 MySQL 遇到了 Bug 或硬件故障，你一般都无须做超乎寻常的事情，甚至包括预防服务器断电。InnoDB 在启动时会执行一个通常的还原过程，将一切都恢复正常。在它的日志文件里，你会看到像下面这样的消息：

```
InnoDB: Doing recovery: scanned up to log sequence number 0 40817239
InnoDB: Starting an apply batch of log records to the database...
```

InnoDB 会在日志文件里以百分比的形式打印出还原进程的进度消息。也有些人报告说没看到这些消息，直到整个过程完成之后。请耐心等待，没有办法能加快这个过程。把它关闭再重启只会使这个过程更加漫长。

当你的硬件出现一个严重的问题（例如内存或磁盘错误）时，或者在 MySQL 或 InnoDB 里碰到了 Bug 时，你就不得不进行干预，或者强制还原，或者防止从那个错误处开始还原。

InnoDB 损坏的根源

InnoDB 通常情况下是相当健壮的。它是建立在可靠性之上的，有许多内建的合理性检验来防止、发现和修复被损坏的数据——比其他存储引擎要多得多。然而，它也不可能让自己“刀枪不入”。

从最低限度上来讲，InnoDB 依赖于无缓冲 I/O 调用和 `fsync()` 调用，在数据被安全写入物理媒体之前，它们都不会返回。如果你的硬件没有真正写入这些数据，那 InnoDB 也无法保护数据。一次服务器崩溃就会引起数据损坏。

许多 InnoDB 损坏问题都是与硬件相关连的（例如在损坏的页上写入是由掉电或坏内存引发的）。然而，在我们的经历里，配置不当的硬件也是一个很大的问题来源。常见的配置不当包括在 RAID 卡里开启了回写 (Writeback) 缓存，而 RAID 卡又没有后备电源；或者在硬盘驱动器上开启了回写缓存。这些错误都会使控制器或驱动器谎报 `fsync()` 已经完成，实际上，那些数据还只在回写缓存里，而不是在磁盘上。换句话说，硬件无法提供 InnoDB 所需要的保证，来保障数据的安全。

有时，这些不当的配置都是机器默认配置好的，因为这有更好的性能表现——这对一些用途而言确实是好的，但是对事务数据库服务器来说却不好。如果服务器不是你设置的，你最好经常地检查一下这些配置。

如果你在网络附属存储 (Network-attached Storage, NAS) 上运行 InnoDB，你也会得到被损坏的数据，因为在该类设备上完成 `fsync()` 操作，只是意味着设备已经接收到该数据。这时，如果 InnoDB 崩溃了，那数据还是安全的，但是，如果 NAS 设备崩溃了，那数据就可能没了。

有时数据损坏会比其他问题糟糕好几倍。严重的损坏会使 InnoDB 或 MySQL 崩溃。但是，轻微的损坏仅仅意味着一些事务的丢失，因为日志文件不是真正与磁盘同步的。

怎么还原损坏的 InnoDB 数据

InnoDB 的损坏有 3 种主要类型，每一种对数据还原有着不同层面的影响：

二级索引损坏

通常而言，你可以使用 `OPTIMIZE TABLE` 来修复损坏了的二级索引。同样地，你可以使用 `SELECT INTO OUTFILE`，删除后再重建这张表，然后使用 `LOAD DATA INFILE` 来完成修复。这些修复过程都是通过建立新表，进而重建受到影响的索引。

集群索引损坏

你可能要用到 `innodb_force_recovery` 设置来导出这个表（更多内容在后面展开）。有时，这个导出过程会使 InnoDB 崩溃，如果确实崩溃了，那你就需要导出某个范围内的行，跳过那个会引起崩溃的页。一个损坏了的集群索引要比一个损坏的二级索引严重得多，前者会影响到数据所在的行，但是，在许多情况下，修复这些受影响的表也是有可能的。

损坏的系统结构

这里指的系统结构包括了 InnoDB 事务日志、表空间里的撤销日志区域、数据目录。这类损坏可能需要做整个数据库的导出和还原，因为这些结构影响到了 MySQL 内部绝大部分的工作任务。

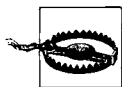
你一般可以修复一个损坏的二级索引而不丢失数据。然而，另有两种情形会引起一些数据的丢失。如果你已经有了一个备份，那最好还是从备份记录里还原数据，而不要试着从损坏的文件里去提取数据。

如果你必须从损坏的文件里提取数据，那一般要让 InnoDB 运行起来，然后使用 `SELECT INTO OUTFILE` 导出数据。如果服务器已经崩溃了，无法再启动 InnoDB，你可以配置 InnoDB，防止它做常规还原，并阻止其后台处理过程的运行。这也许可以帮你启动服务器，在缺少检查或没有检查的情况下做一个逻辑备份出来。

`InnoDB_force_recovery` 参数用来控制 InnoDB 在启动和做常规操作时要做哪一种类型的操作。通常情况下这个值是 0，最高可以提高到 6。MySQL 使用手册里记录了每个数值会产生的操作行为。在这里，我们不再重复这段信息，但是，我们要告诉你：在有点危险的前提下，你可以把这个数值调高到 4。使用这个设置时，若数据页有损坏，你就会丢失一些数据；如果将数值设得更高一点，你会从损坏的页里提取到坏掉的数据，但是提高了执行 `SELECT INTO OUTFILE` 的崩溃风险。换句话说，层次 4 对你的数据没有损害，但会丧失修复问题的机会；层次 5 和层次 6 会更主动地修复问题，但是损害数据的风险会很大。

510 当你把 `innodb_force_recovery` 设为大于 0 的某个值时，InnoDB 在本质上就是只读的，但是你仍然可以创建和删除表。这可以阻止进一步的损害，InnoDB 会放松一些常规检查，因为当发现坏数据时，它不会特意崩溃。在常规操作中，这就是一个安全保障。但是当你在还原时，就不会希望这样了。如果你需要强迫 InnoDB 还原，有个好主意就是配置 MySQL，使它在操作完成之前，不允许接受常规的连接请求。

如果 InnoDB 的数据损坏到了根本不能启动 MySQL 的程度，你可以使用 InnoDB 还原工具箱从表空间的数据页里直接取出数据。这些工具由本书的几个作者开发，可以在 <http://code.google.com/p/innodb-tools/> 网站上免费获取。



警告：我们通常不会提到 MySQL 中具体的错误，但是在许多版本的 MySQL 里，在定义 `innodb_force_recovery` 之后，会有一个非常严重的 Bug 阻止你执行还原过程。您可以在 <http://bugs.mysql.com/28604> 上追踪到这个 Bug 的状态。如果你在导出一个损坏了的 InnoDB 表时看到 “Incorrect key file” 错误信息，你就需要阅读一下这个 Bug 的报告，看看是否确为这个原因。如果是，可以使用 MySQL 5.0.22 来导出数据。作为一个美好愿望，希望你永远都不需要担心此类问题。

11.6 备份和还原的速度

Backup and Recovery Speed

在正确性之后，速度就是高性能系统备份/还原里的最重要因素了。以下是要考虑的与速度有关的因素：

锁定时间

在备份的时候，你的锁会保持多长时间，例如全局的 `FLUSHTABLESWITHREAD` 锁？

备份时间

把备份复制到目标位置上要多长时间？

备份的负载

当备份复制到目标位置时会给服务器的性能带来多大的影响？

还原时间

还原时间包括了把备份记录从存放位置复制到 MySQL 服务器的时间、重放二进制日志的时间，以及其他相关时间。

这里最大的权衡在于备份时间和备份负载。通常，你可以用一项开销来抵消另一项开销，例如，你可以以牺牲更多服务器性能的代价来优先做备份。

你可以设计备份以利用负载模式的优点。例如，如果服务器在夜间有 8 小时处于 50% 负载量，那你可以试着把备份方案设计为给服务器增加少于 50% 的加载，但仍然保证在 8 小时内完成。

你可以通过多种方法来达到这个效果：例如，你可以用 `ionice` 和 `nice` 优先执行复制和压缩操作，使用不同级别的压缩方式，或者在备份服务器上压缩数据，而不是在 MySQL 服务器上。你也可以是用 `O_DIRECT` 或 `madvise` 使复制操作绕过操作系统的缓存，这样它们就不会去“污染”服务器的缓存了。

通常来讲，从裸备份里获取数据的速度会更快一点，相关工作也更少。然而，逻辑备份是一个重要的补充，因为裸文件不够轻便，也不是永远都可以还原的，而且当其中有损坏时也难以被发现。如果定期从裸文件副本里做逻辑备份，你花一点额外的精力就可以获得两者之长。

11.7 备份工具

Backup Tools

除停止服务器、还原数据、重启服务器之外的那些事情都是相当复杂的，这是演练和脚本化这些操作的根本原因。在本小节接下来的篇幅里，我们要介绍一些在备份/还原时很有用的工具，它们可以用来生成操作脚本，也可以用来做进一步到位（One-shot）的导出和还原。

本书的第一版里说到过“如果你有复杂的配置，或者有不同寻常的需要，可能没有一个方案能为你完成预定的任务。这时，你需要构建一个自定义方案了。”时间已经过去了很久，今天我们建议不要使用脚本来操作备份工具，除非真地必须。如果某个现存的工具能很好地满足你的需要，那真是个不错的机会；如果没有，你也能修改其中的一个来完成你要做的事情。

尽管如此，有一些更复杂的备份场景还是要自定义的脚本来做，因此，我们在本章末尾部分包含了一些基本的 how-to 建议。

11.7.1 Mysqldump

mysqldump

用来创建数据和式样逻辑备份的工具，最流行的应该是 `mysqldump` 了。`Mysqldump` 是随着 MySQL 服务器一起提供的，所以，你一般都用不着安装它。它是个多用途工具，可以用来完成很多任务，例如将表从一台服务器复制到另外一台：

```
$ mysqldump --host=server1 test t1 | mysql --host=server2 test
```

在本章里，我们已经展示过好几个使用 `mysqldump` 创建逻辑备份的例子了。在默认条件下，它输出的是一个包含了所有建表和填充数据命令的脚本。它也有一些选项用来输出视图、存储过程和触发器。以下是一些典型用途的例子：

- 将服务器上的所有东西都备份到一个单独的文件里：

```
$ mysqldump --all-databases > dump.sql
```

- 只将 Sakila 示例数据库做逻辑备份：
\$ `mysqldump --databases sakila > dump.sql`
- 只将 sakila.actor 表做逻辑备份：
\$ `mysqldump sakila actor > dump.sql`

你可以用 `--result-file` 选项来指定一个输出文件，这个选项能帮你防止在 Winodwn 上出现换行转换：

```
$ mysqldump sakila actor --result-file=dump.sql
```

`mysqldump` 的默认选项并不对所有有重要目的的备份都适合。你可能想通过指定一些选项来明确地控制输出结果。这里有一些常用的选项，可以使 `mysqldump` 更加有效率，输出的结果更容易使用：

`--opt`

启用一组选项，它们会关闭缓冲区（这个会使服务器耗尽内存），导出数据时把更多的数据写在更少的 SQL 语句里，这样在加载的时候就更有效率了，同时可以做其他一些有用的事情。更多细节可以阅读版本帮助文件。如果关闭了这组选项，`mysqldump` 会在把表写到磁盘之前，把它们都导出到内存里，这对于大型的表而言是不切实际的。

`--allow-keywords, --quote-names`

使用户能在导出和恢复表时，使用保留字作为表的名字。

`--complete-insert`

使用户能在带有不同字段的表之间移动数据。

`--tz-utc`

使用户能在不同时区的服务器之间移动数据。

`--lock-all-tables`

使用 `FLUSH TABLES WITH READ LOCK` 获取一个全局一致的备份。

`--tab`

使用 `SELECT INTO OUTFILE` 导出数据，使备份和还原速度更快。

`--skip-extended-insert`

使每一行数据都有自己的 `INSERT` 语句。必要时这可以帮你有选择地还原某些行。它的代价是大文件导入到 MySQL 时，开销会更大。因此，只有在你需要时，才能启用它。

如果你在 `mysqldump` 上使用 `-databases` 或 `--all-databases` 选项，那么最终导出的数据会是每个数据库里都一致的，因为 `mysqldump` 会同一时间锁定并导出一个数据库里的所有表。然而，来自不同数据库的各个表就未必是相互一致的。使用 `--lock-all-tables` 选项可以解决这个问题。

11.7.2 mysqlhotcopy

`MySQLhotcopy` 是一个 Perl 脚本，包含在标准版 MySQL 服务器下载包里。它是为 MyISAM 表设计的，在我们的可选范围里，它不会做“热”备份，因为它在复制表的时候要锁定所有表。虽然它曾是在活动服务器上做备份的最流行的工具之一，但是如今它已经不怎么流行了。许多高性能安装包正在远离 MyISAM，甚至于你能使用的只有 MyISAM 表，而文件系统快照常缺少插入性，因为它们锁定数据的时间可以更短。

作为一个例子，我们用所有的 MyISAM 表创建了 Sakila 示例数据库的一个副本。为了将这个数据库备份到/tmp，我们运行如下命令：

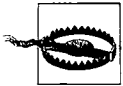
```
$ mysqlhotcopy sakila_myisam /tmp
```

这个命令在/tmp 下创建了一个名叫 sakila_myisam 的子目录，里面包含了从那个数据库里复制出来的所有表：

```
$ ls -l /tmp/sakila_myisam/
total 3632
-rw-rw---- 1 mysql mysql 8694 2007-09-28 09:57 actor.frm
-rw-rw---- 1 mysql mysql 5016 2007-09-28 09:57 actor.MYD
-rw-rw---- 1 mysql mysql 7168 2007-09-28 09:57 actor.MYI
... omitted ...
-rw-rw---- 1 mysql mysql 8708 2007-09-28 09:57 store.frm
-rw-rw---- 1 mysql mysql 18 2007-09-28 09:57 store.MYD
-rw-rw---- 1 mysql mysql 4096 2007-09-28 09:57 store.MYI
```

它复制了数据库里每一个表的数据、索引和表定义文件。为了节省空间，你可以使用 `noindices` 选项让它对每个 .MYI 文件（这个文件是 MySQL 用来重建索引的）只备份 2048 个字节数的数据。如果你使用了这个选项，就要在还原文件之后重建相关表的索引。你可以使用带 `recover` 选项的 `myisamchk` 来做，或者使用 `REPAIR TABLE` 这个 SQL 命令。

Mysqldhotcopy 相当复杂，并不是所有人都喜欢这种兼容性，因此，许多人运行自己编写的脚本，用一种稍微有所差别的方式，来完成本质上完全相同的工作。



警告：用 `innodb_file_per_table` 配置 InnoDB 后，`mysqlhotcopy` 会复制 .ibd 数据文件，但这是没有用的。不要相信这个虚假的安全感，这不是安全备份 InnoDB 数据的途径。

11.7.3 InnoDB 热备份

InnoDB Hot Backup

InnoDB 热备份——`ibbackup`，是一个由 InnoDB (Innobase) 发布的商业化工具。使用它时，不用停止 MySQL，设置锁，或者中断常规的数据库活动（虽然它会给服务器添加一些额外的负载）。它也可以压缩备份记录。

配置 `ibbackup` 的方法是给它提供一个与服务器上 `my.cnf` 文件相匹配的配置文件，并且放在不同的数据目录下。这个工具会读取这两个配置文件，然后把 InnoDB 文件从服务器复制到第 2 个配置文件指定的位置上：

```
$ ibbackup /etc/my.cnf /etc/ibbackup.cnf
```

还原备份记录时，先关闭 MySQL，然后运行下面这个命令：

```
$ ibbackup --restore /etc/ibbackup.cnf
```

这里会有一个小问题：`ibbackup` 只复制了 InnoDB 文件，不包括表的定义或其他必要的文件。Innobase 也提供了一个免费的 `innobackup` 助手脚本，它把文件复制、表锁定和 `ibbackup` 封装在一个单独的命令里，这样就能像复制 InnoDB 文件一样，复制表的定义和 MyISAM 文件了。不像 `ibbackup` 那样，这个脚本会中断 MySQL 的常规处理进程，因为它里面使用了 `FLUSH TABLES WITH READ LOCK`。

在我们的可选范围内，使用 LVM 的快照功能备份 InnoDB 比 `ibbackup` 更加便利和有用。LVM 最大便利之一是不需要在文件系统上再给数据做一次复制——你可以创建快照，按你的意愿执行 InnoDB 备份，然后直接把它

发送到备份目的地。

LVM 和 ibbackup 在通常情况下都有不相上下的性能，具体要看如何配置备份，以及是否有写密集的工作负载。在那种情形下，LVM 会有很多写时复制的系统开销。在另一方面，ibbackup 不会随着数据尺寸的增大而呈线性增长，它的工作原理是逐页复制数据文件，然后在复制过去的文件上重放日志文件来“还原”，直到备份过程结束。

11.7.4 mk-parallel-dump

这个工具是 Maatkit (<http://maatkit.sourceforge.net>) 的一部分。它能同时执行几个备份操作。

在默认情况下，mk-parallel-dump 的行为就像对 mysqldump 的多线程封装，但是，它也能使用 `SELECT INTO OUTFILE` 来输出 tab 分隔文件。mk-parallel-dump 的默认配置是每个 CPU 一个线程，因此服务器上的 CPU 越多，它工作起来也越快。它也可以按预定尺寸一块一块地备份每一个表——这在还原 InnoDB 表时更加快速，其中的好处是你可以避免还原时出现的巨大事务。巨大的事务会使 InnoDB 的表空间增长到非常庞大的规模，若有错误发生，回滚的时间会延长很多。

这个工具还有一些漂亮的功能，例如能够做增量备份；将一些表组成群放入逻辑备份集合里。基准测试显示：并行地进行逻辑备份可以显著提高整体备份速度。

Maatkit 也包含了 mk-parallel-restore——一个做多线程导入时的伴侣程序。以上两者都善于利用 Unix 的经典技术（例如管道、FIFO 设备）来减少压缩/解压缩文件时对系统性能产生的影响。

11.7.5 Mylvmbackup

Lenz Grimmer 的 mylvmbackup (<http://lenz.homelinux.org/mylvmbackup/>) 是一个 Perl 脚本，在使用 LVM 快照做 MySQL 备份时，可以用它来自动化整个过程。它得到一个全局的读锁，然后创建一个快照，再释放锁。接着使用 tar 压缩数据，删除快照，用备份发生时的时间戳来命名这个 tar 压缩包。

11.7.6 Zmanda Recovery Manager

Zmanda Recovery Manager for MySQL，或者叫 ZRM (<http://www.zmanda.com>)，是我们在这里提到的备份/还原工具里面应用最广泛的。它有免费（GPL）和商业两种许可方式。它的企业版里带有一个管理控制台，提供了一个基于 Web 的图形化界面，可用于配置、备份、校验、还原、报告和调度，也能用于 MySQL 集群的备份。它具备一切通常所见的优点（例如后台支持）。

开源版本在基本功能上没有缩水，但它没有那些额外的好东西，例如基于 Web 的控制台。如果你善于使用命令行，它还是非常可用的。例如，你可以在命令提示符之后，做调度和检查备份。

ZRM 实际上不仅仅是个单一的工具，它更像个“备份协调员”。它在标准的工具和技术之上封装了自己的功能，例如 mysqldump 和 LVM 快照。它用标准的格式存储数据，因此，就没有必要购买专用软件来做还原。它强大功能之一是统一还原机制——无论备份是如何做成的，它都可以用一样的途径来还原。

图 11-2 显示了企业版里的日历界面，它能概览 MySQL 备份记录和日志分析器——Zmanda 称之为“数据库事件显示器”。本质上，它就是一个日志搜索工具，你可以使用普通搜索语法来找出想要找的事件，便于将备份还

原到某个日志事件或某个即时点。

安装和测试 ZRM

Zmanda 的网站上宣称：安装、执行和检验一个备份；设置和校验一个每日计划；执行一次还原、总共需要 15 分钟！作为一个检验，我们在一台运行着 Ubuntu 的笔记本上从头开始安装 ZRM。要下载的 ZRM 包很小，我们使用 `sudo dpkg -i mysql-zrm_1.2.1_all.deb` 将它安装好。这里还有几个先决条件，但是，我们使用 `sudo apt-get -f install` 很容易地就安装好了，整个过程花费不到 1 分钟。

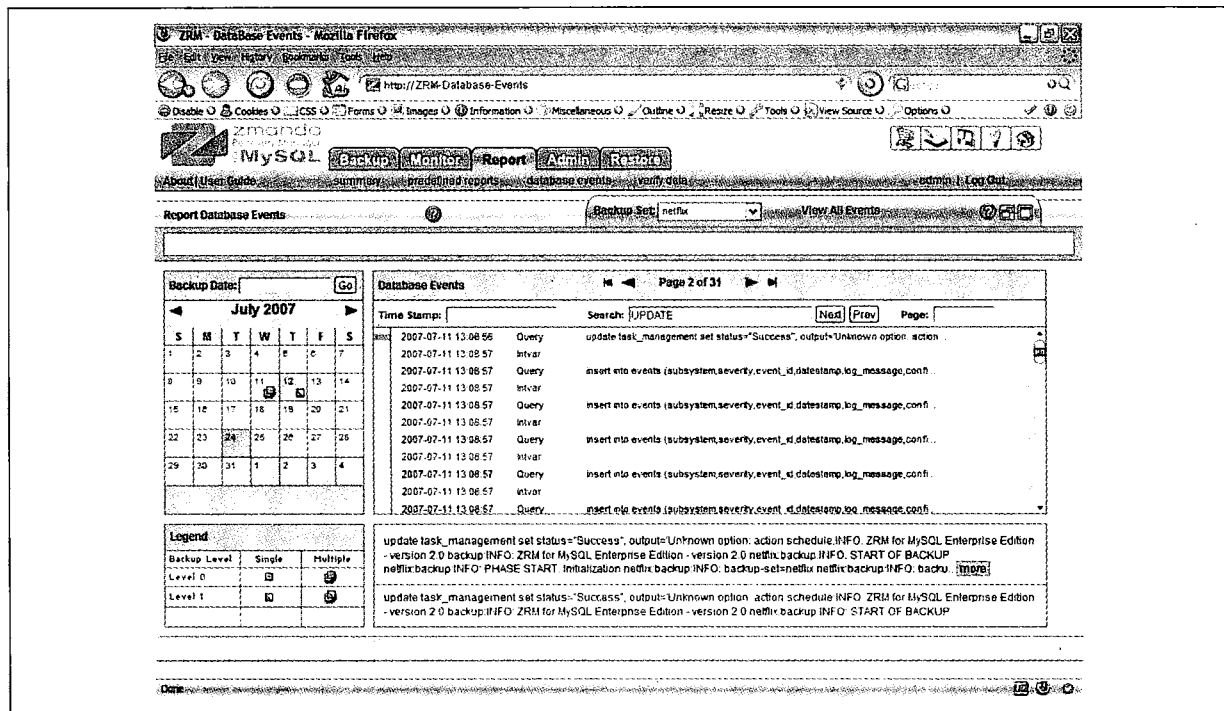


图 11-2: ZRM 的备份日历和日志搜索界面

我们按网站上的指令来配置备份集，它现在是 Sakila 示例数据库的一个逻辑备份。这大概花费了 3 分钟。然后输入以下这个命令来开始备份：

```
# mysql-zrm-scheduler --now --backup-set dailyrun
```

备份过程就只有一会儿，结果文件存放在 `/var/lib/mysql-zrm/dailyrun`。然后，我们再运行一遍备份，并故意给 ZRM 制造出一些错误来，例如杀掉它的一些子进程，给它错误的登录参数。它可以正确地检测到这些错误，然后写入备份通知邮件里发送出去。这里有个要注意的细节是日志会写到系统日志预期的位置上。

最后，我们删除了 `sakila` 数据库，又使用下述命令从最近一次成功的备份记录里将它还原出来：

```
# mysql-zrm-reporter --show restore-info --where backup-set=dailyrun
# mysql-zrm-restore --backup-set dailyrun --source-directory
/var/lib/mysql-zrm/dailyrun/20070930134242/
```

通常而言，ZRM 是一个精心设计的系统，具备了良好的错误检查功能，它能自动化许多繁琐的备份/还原工作，而且，正如其名称所显示的那样，它是自下而上都为还原而设计的。

11.7.7 R1Soft

R1Soft (<http://www.r1soft.com>) 提供了一个名叫 Continuous Data Protection 的产品, 这是个商业软件。它与文件系统快照类似, 只是当把快照复制到另外一台服务器上时, 它只能复制其中更改过的那部分数据。你可以用它把数据回滚到多个过去的版本。

11.7.8 MySQL 在线备份

MySQL Online Backup

MySQL 在线备份不是一个工具, 它只是 MySQL 5.2 (现在还是个 alpha 版) 里开发的一个功能, 可能会在 MySQL 6.0 里正式发布。

这个功能的界面就是一个新的 BACKUP DATABASE 语句, 它能以很高的速度对每个表做快照, 然后写入对应的文件里。它或者使用一个默认的驱动来备份所有类型的存储引擎, 或者为特定的一个存储引擎实现驱动以提高备份的效率。默认的驱动会堵塞其他 SQL 语言的运行, 但是原生驱动在备份时就不会这样。还原功能也包含在这个语句里了。

在本文还在写作的时候, 该项目在 5.2 版本的代码里已经有了一个初步的实现, 一些主要的功能已经实现。但是还有很多功能有待完成, 例如 MyISAM 的原生驱动、跨数据引擎的一致性备份。

在线备份是很值得期待的, 当它完成之后, 可能就是 MySQL 最重要的备份工具之一了。

11.7.9 备份工具的比较

Comparison of Backup Tools

表 11-2 提供了本章讨论到的各种备份工具的快速摘要信息。

表 11-2: 各备份工具的特点

| 特点 | mylvmbackup | mysqldump | mk-parallel- dump | mysqlhotcopy | ibbackup |
|----------|-------------|-----------|-------------------|-----------------|----------|
| 是否有数据块处理 | 可选 | 是 | 是 | 是 | 否 |
| 逻辑备份或裸备份 | 裸备份 | 逻辑备份 | 逻辑备份 | 裸备份 | 裸备份 |
| 数据引擎 | 全部 | 全部 | 全部 | MyISAM/ Archive | InnoDB |
| 速度 | 非常好 | 慢 | 好 | 非常好 | 非常好 |
| 能否远程备份 | 否 | 是 | 是 | 否 | 否 |
| 可用性 | 免费 | 免费 | 免费 | 免费 | 商业 |
| 许可证 | GPL | GPL | GPL | GPL | 私有版权 |

11.8 脚本化备份

Scripting Backups

我们建议如果你有现成的系统可用, 就不要重新发明轮子。但是, 你仍然有必要运行自己的脚本, 或者修改一个已有的脚本。下面是我们在工作中看到过的几种备份配置:

- 把许多服务器上的数据备份到几个备份服务器上，这些备份服务器拥有容量巨大、廉价的硬盘驱动器，没有 RAID。备份脚本会为每个备份分配不同的卷（具体位置要看哪个服务器上有足够的存储空间），它也要保证不同的备份生成到不同的服务器上，这样一来，即使丢了任何一台服务器也不是什么大问题。
- 把备份存档记录切分成很多份，对它们进行加密，然后存储到外部的数据中心——像 Amazon 的 S3 服务或其他大型存储服务。
- 把还原与复制集成起来，这样你就能从备份记录里克隆出一个从服务器。

我们展现一个示例程序如下，它带有一些必不可少的占有页面大量篇幅的脚手架代码，我们将列出这样一个典型的备份脚本的各组成部分，并用 Perl 脚本来显示其中的代码片段。你可以把它们当作基石，用它们来创建你自己的脚本。我们将按你使用它们时的大致次序来罗列：

健全性检查

让你和你同事的人生更简单一些——打开严格的错误检查功能，使用英文变量名：

```
use strict;
use warnings FATAL => 'all';
use English qw(-no_match_vars);
```

如果脚本在 Bash 上，你可以启用更严格的变量检查。当有个未定义的变量用于替换，或者程序退出产生错误时，运行以下语句会出现一个错误信息：

```
set -u;
set -e;
```

命令行参数

每个脚本都要接受命令行参数。如果你发现已经把诸如用户、密码之类的信息硬编码在配置里了，那你真地应该将它提升到更高的层次。

```
use Getopt::Long;
Getopt::Long::Configure('no_ignore_case', 'bundling');
GetOptions( .... );
```

连接到 MySQL

标准的 Perl DBI 库几乎到处都有，它提供许多强大而富有兼容性的功能。关于使用它的细节请阅读 Perldoc（可以在 <http://search.cpan.org> 上获得）。

```
use DBI;
$dbh = DBI->connect(
    'DBI:mysql;host=localhost', 'user', 'pass', {RaiseError => 1});
```

关于命令行脚本，请阅读标准 mysql 程序的 -help 显示的文本，其中有大量选项可增进脚本的友好度。举例来说，以下是 Bash 环境下的枚举数据库列表：

```
for DB in `mysql --skip-column-names --silent --execute 'SHOW DATABASES'`
do
    echo $DB
done
```

停止和启动 MySQL

停止和启动 MySQL 的最好方法是使用操作系统上的方法，例如运行 /etc/init.d/mysql 下的初始化脚本。如果在 Windows 平台上，可以使用管理工具里的“服务”。这也不是唯一的途径。你可以用 Perl 通过一个已存在的数据库连接来关闭数据库。

```
$dbh->func("shutdown", 'admin');
```

当这行命令完成后，你也不能确信 MySQL 已经被关闭了，它也许只在进程表里被关闭了。有时，你可以通过下面这行命令来关闭 MySQL：

```
$ mysqladmin shutdown
```

取得数据库和表的目录

每一个备份脚本都会向 MySQL 请求数据库和表的目录。要知道每一个数据库条目并不是真正的数据库，例如有些日志文件系统里的 lost+found 目录和 INFORMATION_SCHEMA。你也要确认脚本能够处理视图。如果 InnoDB 里有很多数据，执行 SHOW TABLE STATUS 会耗费很长一段时间：

```
mysql> SHOW DATABASES;
mysql> SHOW /*!50002 FULL*/ TABLES FROM <database>;
mysql> SHOW TABLE STATUS FROM <database>;
```

表的锁定、缓存刷新和解锁

你一定会需要锁定一个或多个表，或者根据表名锁定需要的表，或者干脆全局性地锁定所有对象；有时还要对它们的缓存进行刷新：

```
mysql> LOCK TABLES <database.table> READ [, ...];
mysql> FLUSH TABLES;
mysql> FLUSH TABLES <database.table> [, ...];
mysql> FLUSH TABLES WITH READ LOCK;
mysql> UNLOCK TABLES;
```

当你在锁定表并取得目录时，要当心竞争条件的发生。新表可以被创建，或者现存的表可以被删除或重命名。如果每锁定一个表就做备份，那你就无法得到一个一致性的备份。

刷新二进制日志

要服务器创建一个新的日志会比较难（这要在锁定表之后，开始备份之前来做）：

```
mysql> FLUSH LOGS;
```

如果你不考虑从日志文件的中间开始执行，这会使还原和增量备份做起更容易。这样做也有一些副作用：在刷新和重新打开日志时，会潜在破坏旧的日志条目。因此，你要小心，别丢掉你需要的数据。

获取二进制日志的位置

你的脚本应该能获得和记录主服务器和从服务器的状态——哪怕服务器只是一台主服务器，或者只是一台从服务器：

```
mysql> SHOW MASTER STATUS;
mysql> SHOW SLAVE STATUS;
```

这两条语句都能显示内容，并忽略你得到的任何错误，因此，你的脚本有可能读到所有信息。

导出数据

你有两个最佳的选择：使用 mysqldump，或者使用 SELECT INTO OUTFILE。

复制数据

使用本章里我们提到过的方法里的一种就可以。

以上这些是所有备份脚本的基石。其中，最困难的部分是编写还原的脚本。如果你想获得一些这方面的灵感，可以去看 ZRM 的源代码。它的脚本能做一些灵活的事情，例如把元数据与备份保存在一起，使还原更容易操作。

保证 MySQL 安全是保护数据完整性和私密性的关键。就像保护 Unix 或 Windows 的登录账号一样，你要确保 MySQL 账号都有足够强度的密码和适当的权限。因为 MySQL 经常是放在网络上使用的，所以，你也要确保 MySQL 所在主机的安全性：谁能访问到它？如果有人嗅探你的网络数据流，他能得到什么信息？

MySQL 具备了一套非标准的安全和权限系统，它能帮你完成很多特定的任务。它基于一套简单的规则来实现，但是，仍然有很多复杂的例外和特殊案例要处理，因此，会显得有点难以理解。在本章里，我们将先看一下 MySQL 的许可机制是如何工作的，告诉你怎么控制哪些人来访问你的数据。因为在 MySQL 的帮助手册里有完整的关于权限的文档可供查阅，因此，我们在此就只解释那些难以理解的概念，并教你如何去做那些看似难以掌握的普通任务。然后，我们会讲到一些基本的操作系统和网络的安全措施，你可以用它们把“坏家伙”阻挡在数据库之外。最后，我们将讨论一下加密，以及如何让 MySQL 运行在高限制条件的环境下。

12.1 术语

Terminology

在正式开始之前，让我们把一些容易混淆的术语定义一下。在本章里，我们用它们来指代那些特定的事情：

认证

你是谁？MySQL 使用用户名、密码和来源主机来认证你的身份。这是赋予你具体权限的先决条件。

授权

你被允许做些什么事情？例如你要有 SHUTDOWN 权限，才能关闭服务器。在 MySQL 里，授予的是全局性的权限，无法与任何特定对象相关联（例如表或者数据库）。

访问控制

哪些数据允许你查看和/或控制？当你试着读取或更新数据时，MySQL 会检查你是否已被授权查看或更改当前选择的列。跟全局权限不一样的是访问控制应用到特定的对象，例如特定的数据库、表和列。

权限和许可

这两个术语大致上指同一个事情，一个权限或许可就是 MySQL 表示的一项授权或访问许可。

12.2 账号的基本知识

Account Basics

MySQL 账号跟大多数系统里的不太一样，因为 MySQL 认为登录的来源也是认证的一部分，相反，Unix 登录常常是只验证用户名和密码。换句话说，Unix 账号的主键是用户名，而 MySQL 的主键是用户名和来源（常常是主机名、IP 地址或通配符）的组合。

如同我们看到的那样，与位置相关的账号会给原本简单的系统增加复杂性。用户 joe 从 *joe.example.com* 登录到系统的情况会跟他从 *sally.example.com* 登录到系统的不一样。从 MySQL 角度来看，他们是完全不同的用户，有着不同的密码和权限；从另一方面来讲，他们又是相同的用户。具体要看你是怎么配置账号的。

12.2.1 权限

Privileges

MySQL 使用你的帐户信息（用户名、密码和位置）来验证你的身份。一旦验证通过，它就要决定你被允许做的事情有哪些。这时，它就会向你的**权限系统**咨询，权限通常是在执行一条被允许的 SQL 查询语句后取得的。例如，你需要有权限表的 SELECT 权限，从里面取出权限数据。

权限也分为两种：与对象(例如表、数据库和视图)相关的和与对象无关的。**对象指定权限**赋予你访问特定对象的权限。例如，它们可以控制你是否能够从表里获取数据、是否可以更改表结构、是否可以在数据库里创建视图、是否能够创建触发器等。MySQL 5.0 及更新的版本有许多额外的对象相关权限，因为它引入了视图、存储过程及其他新的功能。

另外一方面，**全局权限**可以让你执行一些功能，例如关闭服务器、执行 FLUSH 命令、执行各种 SHOW 命令和查看其他用户的查询情况；基于对象的权限能让你操作服务器里的内容（虽然两者之间的差异并非像定义的那样严格）。每一个全局权限都有着联系广泛的安全含义，所以，将它们授予用户时要极其谨慎。

523 MySQL 的权限是布尔型的：一个权限要么被赋予，要么没有。不像其他数据库系统，MySQL 没有明确拒绝权限的概念。撤销一项权限不是去禁止用户执行某个操作，而是把权限从用户那里移除了——如果原来有这项权限的话。

MySQL 的权限也是层次化的，但是有一点别扭。我们将会说明这一点的。

12.2.2 授权表

The Grant Tables

MySQL 使用了一整套的**授权表**来保存用户和他们的权限。这些表都是 mysql 数据库里的普通的 MyISAM 表(注 1)。将安全信息存储在授权表里意义重大，但是这也意味如果服务器配置不当，任何用户都能通过调整这些数据来修改安全设置了。

MySQL 的授权表是整个安全系统的核心。MySQL 提供给你 GRANT、REVOKE 和 DROP USER 的权限（关于这一点，我们稍后继续讨论），使你几乎可以完全控制所有的安全设置。然而，操作授权表一般就意味着执行某些

注 1：它们必须使用 MyISAM 表，千万不要把它们存储引擎改成其他类别的。

任务。举例来说，在旧版本的 MySQL 里，想要彻底删除一个用户的唯一方法是从 `user` 表里 `DELETE` 该用户，然后执行 `FLUSH PRIVILEGES`。

我们不建议你直接修改授权表，但是，你仍然应该知道它们是怎么工作的，这样就可以调试那些意外行为了。我们鼓励你使用 `DESCRIBE` 或 `SHOW CREATE TABLE` 去检查授权表的结构，特别是当你刚使用 `GRANT` 和 `REVOKE` 修改了权限之后。跟仅仅使用它们相比，阅读它们会让你获得更多的知识。

以下是所有的授权表，显示的次序就是当 MySQL 检查一个验证通过的用户是否允许使用某个操作时的查询顺序：

User

每一行就是一个用户账号（用户名、主机名和加密后的密码）以及用户的全部权限。MySQL5.0 又增加一个可选的用户限制项，例如该用户可以保持的最大连接数。

Db

每一行包含了某个用户在数据库级权限。

Host

每一行包含了用户从指定主机登录过来时它在一个数据库里的所有权限。当检查数据库层面的访问时，这些条目会与 `db` 表里的条目“合并”起来使用。虽然它是作为授权表罗列出来的，但是你无法使用 `GRANT`、`REVOKE` 等命令修改这个主机表，你只能手动增加和删除其中的条目。

我们建议你不要动用这张表，以防止造成 MySQL 的维护问题和各种怪异行为。

tables_priv

每一行包含了指定用户和表的表级别上的权限，也包括了视图的权限在内。

columns_priv

每一行包含了指定用户和列的列级别上的权限。

procs_priv (MySQL 5.0 里新引入的)

每一行包含了指定用户和存储程序（存储过程和函数）的权限。

12.2.3 如何检查权限

How MySQL Checks Privileges

MySQL 在授权表里检查权限的次序就是我们在上一节里列出的那样。服务器会在找到匹配的功能授权后停止检查。这就是说，如果它在 `db` 表里找到了与当前要使用的权限相匹配的条目时，就不会再去查询 `process_priv` 表。图 12-1 描述了这个过程。

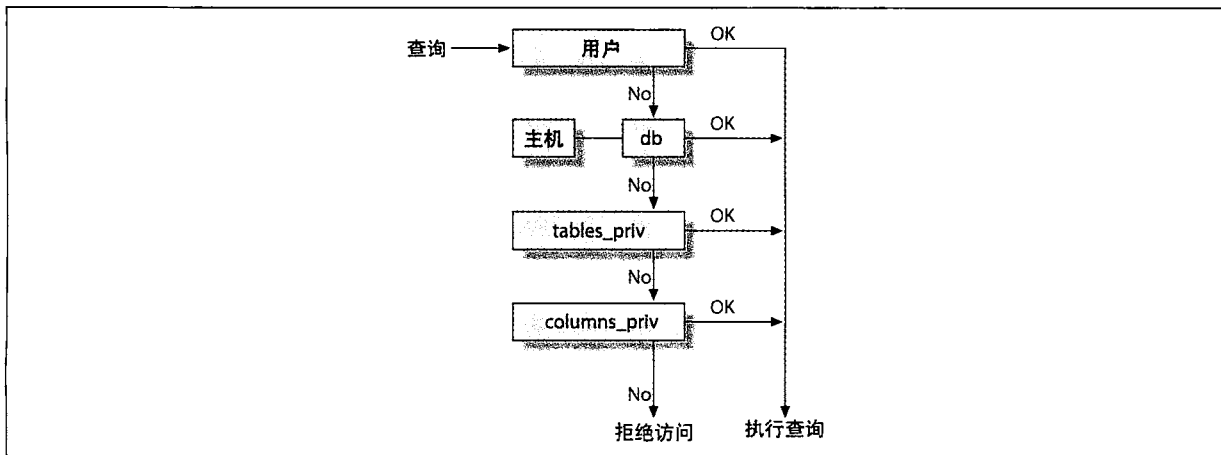


图 12-1: MySQL 是如何检查权限的

525

MySQL 相当于是在缓存的授权表里做 SELECT 操作来决定采用哪个权限。这个虚拟语句里的 WHERE 子句包含着每个表里的主键字段。有些列允许模式匹配，它们中的大部分在有特殊的列值时，还具有“神奇的”含义，例如当它们为空的时候。更多的细节可以查阅 MySQL 使用手册。

你大概会花费大量时间在学习授权表和它们的工作方式上，那些知识可能在偶然间唾手可得。但是，我们不建议你这么去做，除非是必须，与此相比，你更应该阅读下一节内容。只有当你觉得无法（或者说是因为它太复杂而不能够）用 GRANT 和 REVOKE 构建安全系统时，深入钻研授权表才是有价值的。

12.2.4 增加、删除和查看授权

我们建议在 MySQL 里增加用户账号、增加和删除权限都要通过 GRANT 和 REVOKE 命令来做，这两个命令在 MySQL 使用手册里都有完备的文档可供参考。它们提供了一个简单的语法，可做许多种变换，却无须知道底下的授权表和它们不同的匹配规则。你可以使用 GRANT 来增加新用户账号，或增加一项权限；但 REVOKE 只能删除权限，无法删除账号。想删除用户账号，你只能用 DROP USER 来做。

你可以使用 SHOW GRANTS 来查看一个用户的授权情况，其结果的显示内容就是根据该用户当前的权限重新创建用户时所用的语句。例如，下面是默认安装的 Debian 系统里 root 登录后的显示：

```

mysql> SHOW GRANTS;
+-----+
| Grants for root@localhost |
+-----+
| GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost' WITH GRANT OPTION |
+-----+

```

这个语句显示了默认情况下该用户所得到的授权，因此，这也是查看你的登录名和权限的捷径。这里显示的用户拥有了所有的权限，但是没有显示密码，这意味着不用输入密码，这个账户也可以登录。（注 2）这是非常不安全的！当你刚安装完一个全新的 MySQL 后，检查这些安全设置就是你首先要做的事情之一。

注 2：唯一可以减轻问题严重性的因素就是这个用户无法从其他任何一台主机上登录到系统，但是，这不能作为一项安全设置来考量。

如果你想看到其他用户的授权信息，你需要给这个特定的用户指定好用户名和主机名。举例来说，在同一个 Debian 系统里，`user` 表里有下面这样一些条目：

| user | host |
|------------------|-----------|
| repl | % |
| root | 127.0.0.1 |
| root | kanga |
| debian-sys-maint | localhost |
| root | localhost |

要注意到这里有三个 `root` 账号！如果你想查看其中特定一个的授权情况，就必须指定它的用户名和主机名。默认的主机名是%，所以，遗漏主机名会引发一个错误：

```
mysql> SHOW GRANTS FOR root;
ERROR 1141 (42000): There is no such grant defined for user 'root' on host '%'
```

如果你要给一个用户授权而不指定主机名，可以给该用户授权一个 `@'%'`（即任何主机）。

没有什么能阻止你使用标准的 `INSERT`、`UPDATE`、`DELETE` 命令直接修改授权表，但是，坚持使用 `GRANT` 和 `REVOKE` 命令可以把你跟这些表格变化区分。若直接修改这些表，容易犯非常严重的错误。例如，MySQL 不会阻止你把它无法解释的数据写入这些表里去。`GRANT` 和 `REVOKE` 就是被推荐用于权限管理的最好途径，而且一直会这样。

如果你决定要手工修改这些授权表，而不是用 `GRANT` 和 `REVOKE` 命令，你必须通过执行 `FLUSH PRIVILEGES` 命令告诉 MySQL 你已经这么做了，MySQL 就会从表里重新获取和缓存账户、权限信息。你使用 `INSERT` 或其他普通命令在授权表里做的更改，要等到你重启服务器或者运行 `FLUSH PRIVILEGES` 之后，才会被 MySQL 知晓。

12.2.5 设置 MySQL 权限

Get on the MySQL Privileges

让我们来看一下如何为一个虚拟的组织——`widgets.example.com` 创建一系列适当的用户和权限。我们假定你已经登录到了一个刚刚安装完成的 MySQL 实例上，而且已经使用 `DROP USER` 把所有默认账户都删除了。你要检查一下 `mysql.user` 表看看是否已经做了这些。

MySQL 不支持你在别的数据服务器上熟识的角色和分组。MySQL 只支持用户。

在这里，基本的习惯用法就是把三条命令合在一起使用：

```
GRANT [privileges] ON [objects] TO [user];
GRANT [privileges] ON [objects] TO [user] IDENTIFIED BY [password];
REVOKE [privileges] ON [objects] FROM [user];
```

以下是你要创建的各种用户类型以及要设置的各种权限的概况：

密码安全

我们特意使用了一个可爱的“p4ssword”来达到说明的目的，但是，在实际情况下，这不是一个好的密码。你不能因为 MySQL 存储的不是明文，而不注意密码的复杂度。任何能连接到你 MySQL 服务器的人都可以启动一次强力攻击来试着找出你的密码，而 MySQL 这边不像其他类型的密码系统（例如 Unix 密码）那样，

能够有许多种常用的手段来检测并防止此类攻击的发生。MySQL 也没有提供任何一种方法让管理员能强制用户使用一种好的密码标准。你不能使用 libcrack 去连接 MySQL，并要求密码都要符合它的标准，不管这种想法有多酷。外面有许多好的工具和网站能帮你和你的用户生成足够强的密码——我们建议你采用其中的一种。

系统管理员账号

在许多大型组织里，你有两个重要的管理员角色。system administrators 管理的是“物理”服务器，包括操作系统、Unix 登录账号等等；database administrators 专注于数据库服务器的管理。你怎么分配管理账号要取决于你的需要——你可能想让事情简单一些，要求任何要做管理任务的人登录到 MySQL 后都是超级用户；或者是你可能想为每一个需要管理访问的人分别创建一个单独的账号。我们也尽量能简单地开始，只创建了一个名叫 root（像传统的 Unix 里的超级用户一样）的超级权限用户：

```
mysql> GRANT ALL PRIVILEGES ON *.* TO 'root'@'localhost'  
-> IDENTIFIED BY 'p4ssword' WITH GRANT OPTION;
```

数据库管理员账号

当有超过一个 DBA 需要访问 MySQL 时，为他们分别创建一个单独的账号要好过让他们共用一个 root 账号。这样的设置提供了更强的责任性和可审查性：

```
mysql> GRANT ALL PRIVILEGES ON *.* TO 'john'@'localhost'  
-> IDENTIFIED BY 'p4ssword' WITH GRANT OPTION;
```

每个雇员一个账号

widgets.example.com 里的典型雇员就是客户服务代表，负责登记来自电话的订单、更新现有的订单等等。就让我们假设 Tera 就是一个客户服务代表，她登录到一个客户应用里，后者会把 Tera 的用户名和密码传送到 MySQL 服务器里验证和授权。创建 Tera 账号的命令如下：

```
mysql> GRANT INSERT,UPDATE PRIVILEGES ON widgets.orders  
-> TO 'tera'@'%.widgets.example.com'  
-> IDENTIFIED BY 'p4ssword';
```

Tera 必须提供她的用户名和密码给应用程序，应用程序通过验证后就能让她增加新订单或者更新现有订单，但是，她无法再返回去删除自己刚才创建的订单了。在这个配置里，每一个 *widgets.example.com* 的雇员都是使用他们各自的数据访问权限登录到系统里输入订单。他们不是共享一个“应用程序账号”，而是每个雇员都用自己的用户名和权限对订单做各种操作。

模拟组

MySQL 没有提供用户组或角色这样的功能，但这些在其他数据服务器上是很普遍的。有时，创建一个账号，将它的名称命名在一个特定的雇员或应用角色名称例如 *custserv* 或者 *analyst* 后面，这非常有意义，但在这里我们打算这么做。

日志，只写访问

把 MySQL 用作各种数据类型的日志的后台也很常见。不管你是否正在用 Apache 记录 MySQL 上的每一个请求，或者当你的门铃响起的时候，你就一直在追踪，日志就是一个只有写入的应用，它可能只需要对一个单独的数据库或表进行写入。你可以像下面这样创建一个日志账号：

```
mysql> GRANT INSERT ON logs.* TO 'logger'@'%.widgets.example.com'
-> IDENTIFIED BY 'p4ssword';
```

这条命令会在 user 表里新增加一行，但是，我们没有指定给它全局权限，因此，在这行里的所有权限列都是 N。添加这一行的唯一用途就是让用户在提供了密码的前提下从任何一个主机都可以登录进来。因为我们针对某个数据库指定了一项权限，所以就会有一些有趣的数据保存到 db 表里。在 db 表里，这一行的各个列也都是 N，除 the Insert_priv 列是 Y。

备份

一个备份用户会通过 mysqldump 来做备份，他一般只需要用到 SELECT 和 LOCK TABLES 两项权限。如果用户是使用带-tab 选项的 mysqldump 来做 tab 分界符文件的导出，或者用 SELECT INTO OUTFILE，那么，你还要给这个用户赋予一个 FILE 权限。下面是一个备份用户的示例，他只能在本地主机上登录：

```
mysql> GRANT SELECT, LOCK TABLES, FILE ON *.* TO 'backup'@'localhost'
-> IDENTIFIED BY 'p4ssword';
```

为了保证许多备份操作的一致性，这里还会用到 FLUSH TABLES WITH READ LOCK，这就需要 RELOAD 的权限了。这个权限也允许用户做其他几个常见的操作，例如 FLUSH LOGS。

操作和监控

可能有时你会想让某人或某些东西（例如一个用户，或者网络操作中心里一些监控软件，或者 NOC）来访问 MySQL 服务器，帮你维护系统或者修复故障。这个用户账号要求能够连到数据库，可以使用 KILL 和 SHOW 命令，还可以关闭服务器。因为这些功能都非常强大，所以，需要将它限制在一台单独的主机上使用。这就意味着即使是未授权的用户掌握了密码，他也必须在 NOC 里做这些操作。以下这条语句就可以实现这些要求：

```
mysql> GRANT PROCESS, SHUTDOWN on *.*
-> TO 'noc'@'monitorserver.noc.widgets.example.com'
-> IDENTIFIED BY 'p4ssword';
```

你可能还要把 SUPER 赋予他，这可以让他能执行 SHOW INNODB STATUS。

12.2.6 权限在 MySQL 4.1 里的变化

Changes Introduced in MySQL 4.1

MySQL 4.1 里引入了一个新的更加安全的密码散列方式，但是，你仍然可以在新版本里使用旧的密码散列方式（甚至在 MySQL 5.0 及更新的版本里继续使用）。我们建议您还是少用旧的密码散列方式，因为它很容易被破解。如果你注重安全问题的话，就要使用 MySQL 4.1 或更新的版本，并采用新的散列方式。

有一些 GNU/Linux 分发版为了能和老的客户端程序兼容，就将 MySQL 默认地配置为使用旧的密码散列方式。你要检查一下你的默认配置里是否也选择了旧的密码方式。如果你希望你的 MySQL 服务器能拒绝任何使用不安全的旧风格密码的尝试，你可以在服务器的配置文件里设置 secure_auth 选项。另外，还有一个类似的选项可用于客户端程序，它能阻止它们使用旧格式发送密码给服务器——哪怕是服务器要求它这么做。这是个好主意，因为旧格式很容易被嗅探和破解。

新风格的密码是以星号开始的，这样你就能用目测的方法把它们区别开来。在大多数情况下，从旧 MySQL 版本升级过来的用户账号都能正常验证。但是，当以前的 MySQL 4.1 客户端程序试图用新格式的密码连接到更新的 MySQL 服务器时，将无法实现。为了修补这个问题，你可以手工使用 OLD_PASSWORD() 把账号的密码改回

到旧的散列方式，或者升级客户端程序的 MySQL 客户端库。

12.2.7 权限在 MySQL 5.0 里的变化

Privilege Changes in MySQL 5.0

MySQL 5.0 增加了一些新的权限，也略微改变了一些现有的系统安全行为。这一节对这些变化做一个概述。在你升级到任何新版的 MySQL 之前，应该阅读一下它的发布声明，了解一下其中增加了什么新内容、更改了哪些东西。

530 存储程序

我们在第 5 章里讲到过 MySQL 5.0 增加了对存储程序的支持。这些程序能在两种安全上下文里执行：作为定义者（例如那个定义程序的用户）和作为调用者（例如那个调用程序的用户）。

存储程序一般被用作代理人，把特定的权限赋予那些无法直接访问表的用户。常用的方法是创建一个授权用户，然后，作为定义者创建一些列的存储程序，并给予它们 SQL SECURITY DEFINER 特性。表 12-1 说明了一个存储过程是如何允许用户使用另外一个用户的权限来执行语句的。

表 12-1：一个存储过程里的安全上下文语句

| 调用过程的用户 | 安全上下文语句 | |
|-----------------|-------------------------|--|
| | 使用 SQL SECURITY INVOKER | 使用 SQL SECURITY DEFINER 和 DEFINER=LegalStaff |
| LegalStaff | LegalStaff | LegalStaff |
| HumanResources | HumanResources | LegalStaff |
| CustomerService | CustomerService | LegalStaff |

采用这个方法使你能根据用户的具体身份授予或取消他对表的访问权，同时，当你不希望这个用户直接访问表的时候，就可以授予他在表里执行特定的操作——这些操作就封装在存储过程里。举例来说，你有一些私密的法律数据（例如与某外部当事人签订的合约的履行情况）存放在一组表里，它们只对法务人员可见，但是你的客户服务代表需要能够更新这些表里的某一个特定的列。这时，你要取消除了法律人员之外的其他人在这些表里的 SELECT 权限，然后，编写一个存储过程，让其他人能更新他们需要的字段，并使用 SQL SECURITY DEFINER 让存储过程以 LegalStaff 的权限运行。这跟在 Unix 风格的操作系统里使用的 SUID 权限非常相像。

存储程序的命名空间是每个数据库一个，所以，你可以同时有 db1.func_1() 和 db2.func_1()，而不会引起任何命名冲突。

MySQL 会检查存储程序里每条语句的所需权限，执行该程序的权限不会覆盖到其中的语句里，所以，核准这些语句的执行权限要么是根据定义者，要么是根据调用者，具体要看你创建程序时是使用了 SQL SECURITY DEFINER 还是 SQL SECURITY INVOKER。

触发器

MySQL 5.0 还增加了触发器的支持。如果触发器没有用 SQL SECURITY DEFINER 特性来定义，那么在执行的时候就需要专用的权限。这就会产生令人迷惑的影响，例如在一个表里运行 UPDATE、INSERT 或 DELETE 时，会

得到下面这样的错误信息：

```
mysql> INSERT INTO ...;  
ERROR 1142 (42000): Access denied; you need the SUPER privilege for this operation
```

如果触发器不是使用 SQL SECURITY DEFINER 特性创建的，那么，往表里插入数据的用户必须具有 SUPER 权限才能执行这个触发器，这也是为什么刚才那段错误信息看起来像是在说插入表的操作需要 SUPER 权限。(MySQL 5.1 包含了一个 TRIGGER 权限，它会使错误信息少一些迷惑性。)

MySQL 会检查触发器里语句的所需权限，如同它在存储程序上做的那样。

视图

就像存储过程和触发器一样，你可以使用定义者或调用者的权限来执行视图。定义者权限可以让你访问到视图，但是不能访问视图下面的各个表。

这可以让你实现行级安全，但是也可以限制对列的访问。我们相信这是比使用 GRANT 来指定行级权限更好的解决方案，也更易于维护。如果把视图放在一个单独的数据库里，你可以简单地赋予数据库级的权限给用户就行了，用不着去维护单独的表和视图上的权限。图 12-2 显示了用 GRANT 赋予特定行的访问权和把相关列放入一个视图两种方法之间的差异。

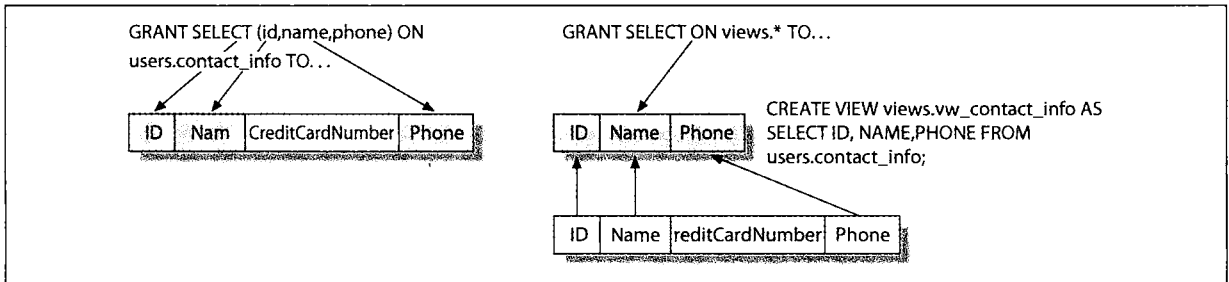


图 12-2：通过定义视图简化对特定列的访问

在图 12-2 的左边，DBA 运行了一条 GRANT 语句，给那些单独行赋予了粒度适当的访问权限。这会减慢所有数据库的访问速度，并要为每一个要求列级许可核查的表单独执行 GRANT 语句。

在图 12-2 的右边，DBA 创建了一个名叫 views 的数据库，用来存放那一系列的视图。然后，他又创建了一个视图，其中包含的是那个他希望用户拥有合适权限粒度的表的各个列。任何这样的视图都可以被存放在 views 数据库里，而单独的 GRANT 语句会使访问授权更有效率。

INFORMATION_SCHEMA 表的权限

官方的 SQL 标准里定义了一整套视图，它们作为一个整体，以名为 INFORMATION_SCHEMA 表而为大家所知，它们可以为你提供数据库、表以及数据库服务器其他部分的信息，MySQL 尽可能地沿袭这些标准。作为努力的结果，服务器能够自动管理这些表的权限，最好不要显式地定义它们的权限。如果某个用户没有适当的权限去访问这些表里的行或者值，MySQL 也就不会显示行给他看，只返回 NULL 形式的结果值。例如，一个用户不能看到 INFORMATION_SCHEMA.TABLES 视图里的表，除非他在这些表上有一定的权限。这也类似于 MySQL 的 SHOW TABLES 的行为：MySQL 不会显示出该用户无权查看的表。

532

12.2.8 权限与性能

Privileges and Performance

权限看上去跟性能没什么太大关系，但是，它们在某些环境下确实会引发性能问题。其中有一些事情需要考虑：

权限太多

如果在授权表里，权限条目过多，它们的开销会很显著。当数据库检查一个用户是否有权限执行当前语句时，它要核对每一条添加进来的权限条目。同时，权限的存储也会消耗内存。

权限的粒度过细

在 MySQL 中如果权限层级里的每一层（用户、数据库、主机、表以及列）都做权限核对显然过于昂贵。核对全局权限相对来说比较快速，但是，即使你仅仅定义一个列的权限，服务器就会潜在地用全局的、数据库的、表的和列的权限去检查每一个查询（回想一下第 524 页的“如何检查权限”，服务器从最高层次开始，持续查找，直到找到跟当前所需权限匹配的授权）。

列的权限和查询缓冲区

在本书写作时，带有列权限的查询访问一个表时，还无法受益于查询缓冲区。像上一节里我们讨论过的那样，我们建议用视图代替列权限，从而避免这里以及其他由列权限带来的问题。

在默认情况下，MySQL 认证用户时，会做前向和逆向的 DNS 查询。在你的 `my.cnf` 表增加一项 `skip_name_resolve` 就可以关闭这个查询。这对于安全和性能两者都很有好处，因为它提高了连接速度，减轻了对 DNS 服务器的依赖，降低了遭受拒绝式服务攻击的可能性。

523 这种改变的副作用是它阻止了你在 Host 列里用主机名定义用户。这类用户定义会停止工作，作为替换方案，你必须使用 IP 地址（你仍然可以使用通配符，例如 `192.%`），也可以使用一个特定的值——`localhost`，哪怕 `skip_name_resolve` 选项已经开启了。

12.2.9 常见问题和解决方案

Common Problems and Solutions

MySQL 帮助手册对权限有着详尽的说明，因此，我们把上一节的篇幅缩减掉了一些，用在本节里来讨论常见的需求、小技巧 and 意外行为，你可以把它用作快速参考或问题排除手册。接下来，我们就开始描述常被问及的问题、常见的任务和我们遇到过的困境。

连接时出错

许多因为连接 MySQL 服务器时遇到麻烦的用户把邮件列表、论坛和 IRC 频道塞得人满为患。对于这些问题有多种原因可以解释，从因为 `my.cnf` 里定义了 `skip_networking` 而导致 TCP 连接失败，到 `bind_address` 把一个不匹配的 IP 地址设在了服务器上，再到错误的 GRANT 语句，一直到防火墙为止。在此，我们无法讲到所有的原因，但是，在 MySQL 帮助手册里有一节是专门讲这个主题的。

通过 localhost 还是 127.0.0.1 来连接

主机名 localhost 一般就是 IP 地址 127.0.0.1 的别名。但是，在 MySQLd 默认行为里，两者的处理方式有轻微的差别。当你把 localhost 作为连接的一项参数时，MySQL 会默认地使用 Unix socket（注 3）去连接，而不是你所期望的 TCP/IP。因而，下面这行命令就是通过一个 Unix socket 连接的：

```
$ mysql --host=localhost
```

这是一个让人有点遗憾的设计决策，因为它不像人们期望的那样运作，要把它改回来也为时已晚，因为这会打破与旧的应用和客户端库的兼容性。如果你想通过 TCP/IP 去连到你正在运行的机器，你有两种选择：指定一个 IP 地址而不是主机名，或者明确地指定通信协议。以下两行命令里的任何一个都能通过 TCP/IP 去连接：

```
$ mysql --host=127.0.0.1
$ mysql --host=localhost --protocol=tcp
```

在一个相关的记录里，如果在建立 SSH 隧道时，试着用 localhost 去连接 TCP 转发端口，你会发现它不会工作。因为必须使用 TCP 去连接端口，所以，你一定要用 IP 地址 127.0.0.1 来代替。我们会在本章的后面讨论 SSH 隧道。

主机名在另一方面也是特殊的：MySQL 不会拿 localhost 作通配符匹配。换句话说，同时指定 user@'%'和 user@localhost 不是多余的。

安全使用临时表

MySQL 对于临时表没有设置特别的权限，除了 CREATE TEMPORARY TABLE。一个临时表一旦被创建，就被加入该用户的标准的表级权限。这意味着一个用户可能可以创建临时表，却没权限增加更多的列、修改表和添加索引（甚至无法 SELECT）。然而，如果把这些权限授予该用户，又会使他危及其他真正的表——这可不是你想要的结果。

这个问题的解决办法是除了在特地为临时表准备的数据库里，其他地方都不接受用户的这些权限：

```
mysql> CREATE DATABASE temp;
mysql> GRANT SELECT, INSERT, UPDATE, DELETE, DROP, ALTER, INDEX,
-> CREATE TEMPORARY TABLES ON temp.* TO analyst@'%';
```

不允许没有密码的访问

MySQL 本身是允许没有密码访问的。一个没有密码的账号在 user 表里的就是 password 列为空字符串的一行记录。你可以通过不带 IDENTIFIED BY 子句的 GRANT 命令来创建这样一个账号。

你无法彻底禁止 MySQL 的无密码访问，但是，如果你能控制用户连接发起的那台机器，你就可以往它的 my.cnf 文件的 [client] 节里加入下面这么一行条目：

```
password
```

这就能让默认读取该配置文件的程序（这包括了所有 MySQL 分发的程序，除非它们已被配置成别的方式。）一直提示用户名和密码。在 MySQL 5.0 和更新的版本里，你可以把服务器的 SQL 模式设为 NO_AUTO_CREATE_USER 来防止创建没有密码的用户，但是，对于一个执着的用户，他还是可能通过变通的方法来达到此目的的。

注 3：localhost 在 Windows 上不显得特殊，但也可以说有点特殊，因为它意味着将通过命名管道去连接。

要记住，在 `mysql.user` 表里用空字符串做密码的用户就是没有密码的用户，没有一个用户会有空密码。

关闭匿名用户

MySQL 也是允许匿名用户的：授权表里 `User` 列为空字符串的条目就是匿名用户使用的权限。对于这些条目要小心，因为 `SHOW GRANTS` 不会显示出这些权限来，我们认为最好还是移除这些条目。你可以运行 MySQL 提供的 `mysql_secure_installation` 程序来做这个。

记得给主机名单独加引号

用户很容易忘记用户名和主机名是要分别加引号的。下面这条命令无法做到它看上去可以做的事情：

```
mysql> GRANT USAGE ON *.* TO 'fred@%';
```

它看起来是创建一个名叫 `fred` 的账号，他可以从任何地址登录进来，但是，事实上，它是创建了一个名叫 `fred@%` 的用户。正确的语法格式要像下面这样（注意，用户和主机名要分开来加引号）：

```
mysql> GRANT USAGE ON *.* TO 'fred'@'%';
```

不要重用用户名

MySQL 认为相同用户名但是不同主机名就是完全不同的用户。这看起来有助于你对同一用户从不同地方来连接时赋予完全不同的权限，但是，根据我们的经验，这样的做法并不是个好主意，其中潜在的混乱和问题远远超过它能带来的好处。跟他们连接后能做什么相比，确保用户名的唯一性，并用主机名来限制用户能从哪里连接进来，会更加简单。举例来说，你可能已经决定所有连接只能来自本地机器，或者本地网络，或者是一个指定的子网。这是一个很合理的安全防范措施，尽管防火墙是更加安全的连接限制方式（我们过一会儿将进一步讨论）。

MySQL 有着许多兼容性措施，但是，这并不意味着你的生活会更加容易，我们认为最好还是让它保持简单。

赋予 SELECT 权限，允许执行 SHOW CREATE TABLE

给用户授予 `SELECT` 权限，让他能执行 `SHOW CREATE TABLE`，显示出重建一个表的 SQL 语句。这通常情况下是个好办法，但是有时也会泄露出敏感信息。最明显的例子就是 MySQL 5.0 里的 `Federated` 表：用户能够看到数据引擎连接到远程服务器时使用的用户名和密码。（MySQL 5.1 增加了一个单独的机制，用于管理 `Federated` 表的远程连接。）

不要授予 mysql 数据库的权限

如果你把 `mysql` 数据库的权限授予了用户，那么，用户就能自己提升自己的权限、查看其他用户的权限（这也就打开了猜密码攻击的方便之门）、甚至是重命名或更改 MySQL 运行时所需要的表。因此，根本不要把这些表的任何访问权授予普通用户——哪怕是只读访问，这意味着下面这行命令就是个坏主意，因为它授予了全局的权限：

```
mysql> GRANT ... ON *.* ...;
```

如果一个用户拥有更新 mysql 数据库里所有表的许可，那他也应该有 GRANT 的选项。否则，他就有可能通过删除行来删除权限，而且无法再加回来。本书的作者之一就曾因此意外地删除了系统里的每一个用户，然后，不得不关闭 MySQL 服务器，再使用带--skip_grant_tables 选项来重启服务器、还原所有用户。

不要随便授予 SUPER 权限

正如你所期望的那样，SUPER 权限可以让用户做超级用户的操作（例如更改服务器上的只读数据），但是，这里也有一个额外的行为：MySQL 会为有 SUPER 权限的用户保留一条连接，哪怕是服务器已经达到了 max_connections 的极限。这可以让你在服务器无法再接受普通的客户端连接时，还能接入并管理服务器。

你应该尽量避免把 SUPER 权限授予太多的用户，但是，当有一些其他的普通用途（例如清除 master 日志）也需要用到它时，就很难把握了。

用通配符数据库授予权限

MySQL 的数据库模式匹配方式让你无法设定“所有数据库都要这样”。这意味着要删除 mysql 数据的权限将是一个枯燥的过程。使用一个良好的命名约定就能够有所帮助：用一个通用的前缀来命名所有数据库，然后就能使用通配符把权限授予所有匹配的数据库。下面就一个例子：

```
mysql> GRANT ... ON `analysis%`.* TO 'analyst' ...;
```

不幸的是，MySQL 并没有真正可用于解决此类问题的方案，虽然使用命名约定也能缓解一些痛苦。值得注意的是在 GRANT 命令里使用时必须给数据库名加上引号来作为一个标识符（带括号的）。

建立共享主机环境也是一种有用的技术。这样的环境可以限制用户使用他的用户名和一个下划线来访问数据。下划线就是一个通配符模式，所以，你必须在 GRANT 语句里将它忽略掉。举例来说，你可以使用下面这样一个命令来为名叫 sunny 的用户建立一个新的主机账号：

```
mysql> GRANT ... ON `sunny\_%`.* TO 'sunny' ...;
```

你不要把 SHOW DATATABLES 权限授予一个共享主机环境。这样可以确保用户看不到他无权访问的数据库——就是说让他们知道得越少越好。

取消特定的权限

如果你授予权限是全局化的，那你就无法用非全局化的方式来取消它们：

```
mysql> GRANT SELECT ON *.* TO 'user';
mysql> SHOW GRANTS FOR user;
+-----+
| Grants for user@% |
+-----+
| GRANT SELECT ON *.* TO 'user'@'%' |
+-----+
mysql> REVOKE SELECT ON sakila.film FROM user;
ERROR 1147 (42000): There is no such grant defined for user 'user' on host '%' on table 'film'
```

这项权限是被全局化授予的，也就只能全局化地取消。如果试着在一个特定表上取消它，那 MySQL 就会产生一个错误：没有表级权限匹配这个指定的标准。

用户甚至在 REVOKE 之后还能连接到数据库

假定你取消了一个用户的所有权限：

```
mysql> REVOKE ALL PRIVILEGES ON...;
```

这个用户还是可以连接进来的，因为 REVOKE 没有删除用户账号，它只删除了权限。你必须使用 DROP USER 彻底删除他的账号（或者，在老版本的 MySQL 里，从 user 表里删除一行）。

如果你仅仅取消了权限，SHOW GRANTS 会显示出该用户仍然具有 USAGE 权限。你无法取消这个权限，因为这个就是“没有权限”的同义词，仅仅意味着他还是能连接到 MySQL。

当无法授予或取消一项权限时

除了 GRANT 选项之外，你必须拥有要授予或取消的那个权限。这项安全措施可以防止用户相互之间逐级提高自己的权限。如果你试着要取消所有权限，那你必须要有 CREATE USER 权限。

不可见的权限

SHOW GRANT 并不能真正显示出一个用户的所有权限：它仅仅是显示出那些明确授予该用户的权限。一个用户也可以拥有别的许可，可能是由于这些许可被授权给了匿名用户。举例来说，默认的 MySQL 安装程序会把 test 数据库和以 test_开头的其他数据库的权限授予每一个用户。首先，让我们以 root 的名义登录，然后创建一个没有权限的用户：

```
mysql> GRANT USAGE ON *.* TO 'restricted'@'%' IDENTIFIED BY 'p4ssword';
mysql> SHOW GRANTS FOR restricted;
+-----+
| Grants for restricted@% |
+-----+
| GRANT USAGE ON *.* TO 'restricted'@'%' IDENTIFIED BY |
| PASSWORD '*544F2E9C6390E7D5A5E0A508679188BBF7467B57' |
+-----+
```

看仔细了，这个用户看上去是只能连接到数据库，不能做其他任何事情。但这还不是整个故事的全部。为了证明这一点，我们只要以这个用户的身份登录，然后运行 SHOW DATABASE 就知道了：

```
$ mysql -u restricted -pp4ssword
mysql> SHOW DATABASES;
+-----+
| Database |
+-----+
| information_schema |
| test |
+-----+
```

这个服务器里也包含了一个 Sakila 示例数据库，这里没有显示出来的原因是该用户没有 SHOW DATABASES 权限，但是 test 已经在列表里了。事实上，如同你会在下面看到的一样，这个新用户拥有这个数据库以及表的所有权限：

```
mysql> USE test;
mysql> SHOW TABLES;
+-----+
| Tables_in_test |
```

```

+-----+
| heartbeat |
+-----+
mysql> SELECT * FROM heartbeat;
+-----+
| id | ts |
+-----+
| 1 | 2007-10-28 21:31:08 |
+-----+

```

这个用户账号只是不能从表里读数据，但是他拥有绝大多数别的权限。实际上，他甚至可以创建一个新的数据库：

```

mysql> CREATE DATABASE test_muah_ha_ha;
Query OK, 1 row affected (0.01 sec)

```

这个问题的罪魁祸首是 mysql.db 里的两行记录：

```

mysql> SELECT * FROM mysql.db\G
***** 1. row *****
      Host: %
      Db: test
      User:
      Select_priv: Y
... 以下省略 ...
***** 2. row *****
      Host: %
      Db: test\_%
      User:
      Select_priv: Y
... 以下省略...

```

539

请注意 User 列是空的，这意味着匿名用户——实际上包括了所有用户拥有这些权限，哪怕是它们没有出现在 SHOW GRANT（注 4）的输出结果里。我讲这个故事的寓意是 SHOW GRANT 无法把所有的一切都显示给你看。有时你仍然要知道怎么去解读授权表里的那些信息。

这也不是用户权限里唯一的奇怪行为。因为主机名和数据库匹配时，首先匹配最具体的名称，不具体的匹配项的权限会被隐藏起来，哪怕它们的名称更加随意。

设想一下下面这样的场景：相比于采纳我们先前建议的命名约定，最小权限原则会使一个懒惰的 DBA 决定采用的逆向方法。他把所有权限都授予一个用户，然后在 mysql 数据库里，把不希望给他的权限都用权限列为 N 的行将它覆盖掉。

```

mysql> GRANT USAGE ON *.* TO 'gotcha'@'%' IDENTIFIED BY 'p4ssword';
mysql> GRANT ALL PRIVILEGES ON `%. cant be empty. * TO 'gotcha'@'%' ;
mysql> INSERT INTO mysql.db(Host, DB, User) VALUES('%', 'mysql', 'gotcha');
mysql> FLUSH PRIVILEGES;

```

因为在 mysql 模式里，更具体的名称优先于%模式，这个懒惰的 DBA 就会考虑：那用户不能拥有像 mysql 数据库里的 SELECT 那样的权限。下面这行命令就说明这个情况：

```

mysql> SELECT * FROM mysql.user;
ERROR 1142 (42000): SELECT command denied to user 'gotcha'@'localhost' for table 'user'

```

注 4：这个例子说明了 MySQL 权限表里的“神奇”值之一。在 User 列里的空字符串代表了一个匿名用户，也就是当你用不存在的用户名连接到服务器时，MySQL 会采用的验证方式，或者意味着这个权限对任何人都可用。

这么做的问题在于权限方案给任何要在将来修改该用户权限的人设置了一个陷阱。在删除 `mysql` 相关的权限时很容易犯错误，它会破坏 `mysql` 数据库权限的严谨性。换句话说，删除权限实际上会授予更多的权限给他！这方案远非它看上去那样聪明，实际上，这非常危险。它也不会 `SHOW GRANT` 时出现，因此很容易被遗漏。

你也可以按照这样的次序，用主机名匹配来玩类似的游戏。举例来说，如果你想允许那个 `gotcha` 用户从除了某特定主机名称以外的任何主机连接到服务器，你不能采用“否定主机名模式”来达到这个目的，因为 `MySQL` 上没这种东西。唯一的途径就是用同样的用户名创建一个用户，然后指定一个要阻止的主机名和一个假密码：

```
mysql> GRANT USAGE ON *.* TO 'gotcha'@'denied.com' IDENTIFIED BY 'b0gus';
```

现在，当 `gotcha` 从这个主机连接到 `MySQL` 时，`MySQL` 会试着用 `user` 表里的 `gotcha@denied.com` 行来验证他，然后因为密码不匹配而拒绝了他的登录。然而，如果有人认为表里的这个条目是错误的，并把它移除了，或者因为性能原因关闭了主机查找功能，或者认为可能有用户危及到了反向 `DNS`，那么，这个“解决方法”就会显得非常危险，以上情况中的任何一种都会使这个用户无法用 `gotcha@%` 登录。

我们建议你避免这类隐藏的权限，转而采用“聪明的方式”，例如我们展示过的那些方案，要多使用符合常识的方法，不要试着用你不需要的权限去做任何花哨的东西，并遵循最佳实践的指引，例如最小权限原则。

废弃的权限

当你删除对象时，`MySQL` 不会去清除那些旧权限。举例来说，让我们假定你运行了下面这样一条命令：

```
mysql> GRANT ALL PRIVILEGES ON my_db.* TO analyst;
```

然后，又运行了这条命令：

```
$ mysqladmin drop my_db
```

如果 `MySQL` 能清除这个表有关的 `GRANT` 那是再好不过了，但是，实际上，那些权限还留在 `db` 表里。如果你随后又用同样的名字创建了一个数据库，而那些权限也仍然存在。这样问题就产生了，你可能都记不起 `analyst` 账号有过哪些权限。

在 `MySQL 5.0` 及更新的版本里，`INFORMATION_SCHEMA` 表能帮你找出那些陈旧的权限。举例来说，你可以使用一个排除联接查询来找到那些引用不存在的数据库的权限：

```
mysql> SELECT d.Host, d.Db, d.User
-> FROM mysql.db AS d
-> LEFT OUTER JOIN INFORMATION_SCHEMA.SCHEMATA AS s
-> ON s.SCHEMA_NAME LIKE d.Db
-> WHERE s.SCHEMA_NAME IS NULL;

+-----+-----+-----+
| Host | Db       | User |
+-----+-----+-----+
| %    | test\_%  |      |
+-----+-----+-----+
```

你可以针对 `INFORMATION_SCHEMA` 的其他表写出类似的查询。在更早的 `MySQL` 版本里，你必须手工去查找那些陈旧的权限，或者写一个脚本为你做这件事情。

`MySQL` 可以让你为不存在的数据库创建数据库级的权限，但是，不会让你为不存在的表创建表级的权限。如果你需要这么做，那你就不得不直接往 `mysql.tables_priv` 表里插入行。

12.3 操作系统安全

Improving System Security

如果一个攻击者取得了服务器的 root 权限，那即使最周全的考虑和最安全的授权表都帮不上你的忙了。有了无限制的访问权，用户可以很方便地把你的数据文件复制到另外一台运行着 MySQL 的机器上。（注 5）这样，攻击者就能很快地得到一份跟你的数据库一模一样的副本。

虽然数据偷窃不是安全预防措施所面临的唯一威胁，一个有创造性的攻击者会认为在几个星期甚至几个月里，持续对你的数据作轻微的修改会更加有效。根据你保留备份的时间长短，以及多久后才发现数据损坏，这样的攻击会极具破坏性。

12.3.1 指导方针

Guidelines

在这里讨论的一般性指导方针不是一个关于系统安全的全面指导。如果你对安全非常重视——你本来就应该这样——我们建议你去阅读 Simson Garfinkel 和其他一些人合写的《Practical Unix and Internet Security》（O'Reilly）。也就是说：这里只是一些关于如何维持数据库服务器良好安全状态的意见：

不要用特权账号运行 MySQL

Unix 上的 root 用户和 Windows 上的系统（管理员）用户对系统有完全的控制权。如果有人在 MySQL 上发现一个安全方面的 Bug，而你又是作为特权账号在运行，那么攻击者就能获得对服务器的广泛的访问权。对于这个问题，安装指示里也写得很清楚，但是值得再重复一遍：创建一个单独的账号（一般就是叫 mysql），专门用来运行 MySQL。

让你的操作系统随时保持更新

所有的操作系统厂商（Microsoft, Sun, Red Hat, Novell 等）在有安全相关的更新可用时，都会通知用户。现在，你要找出厂商的邮件列表，并订阅它，同时，还要特别关注 MySQL 本身的安全列表。对于那些跟数据库直接交互的软件，例如 PHP 或 Perl，你也要留意一下它们的更新。

在数据库主机上限制登录

是不是每个基于 MySQL 开发应用程序的人都需要一个服务器上的账号？当然不！只有系统管理员和数据库管理员需要机器上的账号。所有开发人员只需要能够使用 TCP/IP 远程登录到数据库做一些查询就可以了。

将产品与其他任何东西隔离开

把你的生产环境和开发环境、测试环境隔离开来。最好是使用完全不同的物理服务器。生产服务器的安全和访问需求跟开发服务器的那些设置完全不一样，因此从物理上隔绝它们很有意义。这也可以防止错误发生，使管理和维护更加方便。它要求从一开始就要创建一些适当的过程和工具，例如建立在数据库之间传输数据的途径。

审查你的服务器

许多大型组织都有内部审查员，他们能评估一台服务器的安全性，然后提出一些改善建议。如果你不幸联系不到审查员，那你可以雇佣一个安全咨询师来做这些审查工作。

使用最强大的就意味着管用

你可以采用一些技术，例如 chroot、jail、zone；或者用虚拟服务器，以及其他更多的手段来隔离 MySQL。

注 5：记住：MyISAM 数据文件便于跨越操作系统和 CPU 架构（提供同样的 CPU 浮点数格式）。

把你的备份记录放在另外一台服务器上是一项重要的安全措施。如果有人侵入过你的服务器，你就需要从一个干净的源开始重装操作系统。一旦完成这个步骤后，你就面临着还原所有数据的任务。如果你有时间，你还可以将被侵入服务器上的数据和已知的完好的数据备份相比较，试着找出攻击者是怎么攻入的。

12.4 网络安全

Network Security

隔离你的服务器，让它们难以被访问到，这一直是最好的方法，但是，你总会有一台 MySQL 服务器，它需要被不在同一台主机上的客户端访问到。下面例举一些可以用来限制服务器“曝光”的技术。

哪怕是只在你自己单位的内部网络上使用你的服务器，你也要采取措施让你的数据远离窥探的眼睛。毕竟，公司里的一些最严重的安全威胁实际上都是来自内部。

要牢记记住这里提到的信息只是确保 MySQL 服务器受到良好保护的出发点。市面上有许多优秀的网络安全书籍可供参考，它们包括 Elizabeth D. Zwicky 等人写的《Building Internet Firewalls》和 Craig Hunt(也是来自 O'Reilly)写的《TCP/IP Network Administration》。如果你对网络安全问题非常关切，你就帮自己一个忙，读一本关于此主题的书（在读完本书之后）。

543

如同在操作系统安全里做的那样，让第三方来审查你的网络也很有用，使薄弱点在别人利用之前就把它们找出来。

12.4.1 只有 Localhost 的连接

Localhost-Only Connections

如果你在一个应用里使用了 MySQL，两者位于同一台主机上（通常就是一个小型或中型的网站），那你就有机会禁止任何来自网络的 MySQL 访问了。消除了接受外部连接的需要，就可以减少许多攻击者入侵 MySQL 服务器的途径。

禁止了网络访问也限制了你远程做管理更改(例如增加用户、日志周转等)的能力，因此，你或者需要通过 SSH 登录到 MySQL 服务器，或者安装一个基于 Web 的应用程序，通过它，你可以完成那些更改操作。在一个 Windows 系统上，远程登录的必要条件会有些困难，但是，还有别的远程访问方法可供选择。有一种解决方案就是安装 phpMyAdmin。但是，你要当心，因为众所周知它是有安全隐患的。

skip_networking 选项告诉 MySQL 不要监听任何 TCP socket，但是它仍然会允许 Unix socket 连接。启动不带网络支持的 MySQL 也很简单，只要把下列选项放在 my.cnf 文件的 [mysqld] 里就可以了：

```
[mysqld]
skip_networking
```

skip_networking 选项有一些不方便的副作用：它会阻止你使用诸如 stunnel 这样的工具来做安全的远程连接和复制，它也不让 Java 应用程序连接到数据库（Connector/J 只能通过 TCP/IP 来连接）。有一个变通的方法是像下面这样配置 MySQL：

```
[mysqld]
bind_address=127.0.0.1
```

这就开启了 TCP 连接，但是，只能来自本地机器上，因此就兼顾了安全和便利。有一些流行的 GNU/Linux 发布包已经将此设为默认配置。



提示：一台配置了 `skip_networking` 的 MySQL 从服务器会很有趣。因为它发起连接到主服务器，从服务器仍然可以得到所有数据的更新，但是，因为它不允许有 TCP 连接，你就会有一个更加安全的“备份复制”，它不会被远程污染。你也无法将它用在容错配置里：没有客户端能够连接到它。

12.4.2 防火墙

54

在任何基于网络的服务里，你要只允许认证过的主机才能连接到服务器，这一点是非常重要的。你可以使用 MySQL 的 `GRANT` 命令来限制用户连接过来的主机，但是，多一层保护总归是个不错的主意。使用多重过滤连接的方法，就意味着一个孤立的错误，例如 `GRANT` 命令里的拼写错误，都会使未经认证的主机无法连接到服务器。在网络层上使用防火墙来过滤连接可以给予你额外的安全保障。（注 6）

在许多组织里，网络安全是由其他非开发组的人员来管理的。这有助于进一步减少因为个人的改变而导致服务器暴露的风险。

可供采用的最安全的办法是给一台机器增加防火墙时，先默认地拒绝所有连接，然后再增加规则，给予那些要访问服务器上服务的主机以访问权。对 MySQL 服务器要做的限制是你应该只允许 TCP 的 3306 端口（这是 MySQL 默认的）上的连接，可能还要开通一个远程登录服务，例如 SSH（一般是 TCP 的 22 端口）。

没有默认的路由

要考虑到装备了防火墙的 MySQL 服务器上不能有默认的路由配置。这样的话，即使防火墙的配置被损害，有些人从外部连接到了你的 MySQL，那些数据包也不会发回到他们那里，永远不能离开你的局域网。

让我们假设你的 MySQL 服务器的 IP 是 192.168.1.10，局域网的掩码是 255.255.255.0。在这样一个配置里，任何来自 192.168.0.0/24 的数据包都被认为是“本地”的，因为它可以通过当前网络接口（大概是 `eth0`，或者主机操作系统上类似的东西）直接到达。来自其他地址的网络流量都将不得不定向到一个网关以到达最后的目的地。因为没有配置默认路由，所以，那些数据包都无法找到它们自己的网关，从而无法到达目的地。

如果你必须允许一些选择好的外部主机来访问你那台带防火墙的服务器，那么就给它们添加静态路由。做这个时候你要确保服务器能够响应尽量少的外部主机。

不配置默认路由的措施也不是万无一失的，它更多的是保护你免于因防火墙的配置错误而带来安全威胁，而不是当整个防火墙受到危害的时候。无论如何，它在各个细小方面都是有所帮助的。

注 6：在我们的用意里，防火墙仅仅是一个设备，用于过滤（可能还有路由）流经它的那些网络流量，不管它是一个“真正”的防火墙、路由器，或者是一个老的 486 机器都没关系。

545 12.4.3 在 DMZ 里的 MySQL

对于许多安装要求而言，仅仅给 MySQL 服务器加个防火墙，还是不够安全。如果你的 Web 或者应用服务器之一受到危害了，一个攻击者就能使用这台服务器来直接攻击 MySQL 服务器。一旦攻击者在带有防火墙网络里获得了其中一台电脑的访问权，他也能在相对较少的限制条件下（在许多配置里都是这样）访问到网络里的其他服务器。（注 7）

现在就把 MySQL 服务器移到它们自己单独的网段里，使外界无法访问到，从而提高安全性。举例来说，想象一下有一个局域网，其中有若干台 Web 或者应用服务器和一个防火墙。防火墙的后面是一台或多台 MySQL 服务器，它们位于一个不同的物理网段并且是不同的逻辑子网里。应用服务器已经被限制访问 MySQL 服务器：它的所有流量要先通过防火墙——这个你可以用严格的设置来做到。如果有人获得了应用服务器的访问权，但是，防火墙只允许应用服务器通过 MySQL 服务器的 3306 端口进行相互通信，因此，入侵者无法对 MySQL 上运行的其他服务（例如 SSH）发起攻击。

你可能要把应用服务器放在 DMZ 里，或者放在它们自己单独的 DMZ 里。这是不是做得过头了？可能。作为安全问题里总是会遇到的情况，你也需要从方便性上衡量一下安全措施，然而，在着手做的时候，你应该知道其中包含的风险。

12.4.4 连接加密和隧道技术

在任何你需要通过公共的（如 Internet）或向网络嗅探开放的（如无线网络）网络去访问 MySQL 的时候，你都要考虑到某种形式的加密问题。这样一来就可以给那些企图拦截连接和嗅探或发送欺骗数据的人制造更大的困难。

作为一个额外的好处，许多加密算法的结果都是压缩了的数据流，因此，不仅你的数据更安全，而且带宽的使用效率也会提高很多。

虽然我们讨论的焦点是一个访问 MySQL 服务器的客户端，但是，客户端也可以是另外一台 MySQL 服务器。当使用 MySQL 的内建复制时，这就很常见了：每一个从服务器使用跟普通 MySQL 客户端一样的协议连接到主服务器。

546 虚拟私有网络

如果一个公司在很远的地方有两个或更多的办公室，那就可以使用多种技术在办公室之间架设一个虚拟私有网络（VPN）。一种常见的解决方案是每一间办公室里都有一个外部路由器，它会加密办公室之间的网络通信。在这样一种情势下，就没什么好担心的了，无论连接到办公室的网络是公有的还是私有的，所有的流量都会被加密。

VPN 排除了使用一个 MySQL 专用的解决方案的必要性吗？也不一定。当 VPN 必须因为某个原因而关闭时，若能继续保持 MySQL 网络流量的私密性，那是再好不过了。通过配置 MySQL 使它只接受来自 VPN IP 地址的连

注 7：这也不完全是对的。许多现代网络交换机可以让你在一个物理网络里配置多个虚拟局域网（VLAN）。不在同一个 VLAN 里的机器相互之间不能通信。

接就可以解决该问题：如果 VPN 被关闭，MySQL 就将无法被访问到。

MySQL 里的 SSL

在 4.1 版本里，MySQL 对 Secure Sockets Layer (SSL) 有着原生的支持——同样的技术被用于当你在 Amazon.com 上买书的时候，或者在你喜欢的旅游网站订购机票时，保证你的信用卡号码的安全。特别值得一提的是：MySQL 采用的是可免费使用的 yaSSL 库（或者在更老的版本里是 OpenSSL）。

有些二进制版本的 MySQL 在默认情况下没有开启 SSL。为了核对你的服务器，你只要检查一下 `have_openssl` 变量就可以了：

```
mysql> SHOW VARIABLES LIKE 'have_openssl';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| have_openssl  | NO    |
+-----+-----+
```

如果它显示的是 NO，那你就需要编译你自己的 MySQL 服务器，或者换另外一个版本。如果它显示的是 YES，那么，数据库访问安全的所有新层次都会开放给管理员。至于你怎么使用它们，还是要看你那些特定的应用在安全方面有什么样的需求。

在大多数基本层次上，你可以只允许加密的会话，让 SSL 协议来保护用户的密码。通过 GRANT 命令里的可选项，你也能要求用户通过 SSL 来连接：

```
mysql> GRANT ... IDENTIFIED BY 'p4ssword' REQUIRE SSL;
```

然而，GRANT 无法对客户端所使用的 SSL 证书设置任何限制。只要客户端和 MySQL 服务器能够商定一个 SSL 会话，MySQL 就不会去检查客户端证书的合法性。

你可以使用 REQUIRE x509 选项对客户端证书做最低程度上的检查：

```
mysql> GRANT ... IDENTIFIED BY 'p4ssword' REQUIRE x509;
```

这是要求客户端证书至少能通过 MySQL 服务器上已经设置承认的 CA 的认证。

另有一种提高安全度的方法是只允许一个特定的客户端证书访问数据库。你可以用 REQUIRE SUBJECT 语法来做到：

```
mysql> GRANT ... IDENTIFIED BY 'p4ssword'
-> REQUIRE SUBJECT "/C=US/ST=New York/L=Albany/O=Widgets
Inc./CN=client-ray.example.com/emailAddress=raymond@example.com";
```

可能你并不关心到底使用的是哪个特定的客户端许可证，只要它是你自己组织签发的 CA 证书就行。在这种情况下，你就可以像下面这样使用 REQUIRE ISSUER 语法来实现这个需求：

```
mysql> GRANT ... IDENTIFIED BY 'p4ssword'
-> REQUIRE ISSUER "/C=US/ST=New+20York/L=Albany/O=Widgets
Inc./CN=cacert.example.com/emailAddress=admin@example.com";
```

在最终的认证里，你可以合并这两个子句用来要求签发者和标题都是预定的值。举例来说，你可以要求 Raymond 使用由你组织签发的特定的 CA 证书，就像下面这样：

```
mysql> GRANT ... IDENTIFIED BY 'p4ssword'
-> REQUIRE SUBJECT "/C=US/ST=New York/L=Albany/O=Widgets
Inc./CN=client-ray.example.com/emailAddress=raymond@example.com"
-> AND ISSUER "/C=US/ST=New+20York/L=Albany/O=Widgets
Inc./CN=cacert.example.com/ emailAddress=admin@example.com";
```

另外有一个跟 SSL 相关的次要选项就是 CIPHER 要求选项，它可以让管理者只允许使用了“可信任的”（高强度的）加密方法的密码。SSL 跟密码是相互独立的，如果一个真正的弱密码被用来保护传输的数据，那么，潜在的高强度 SSL 加密方法就会无效。你可以将选择限制在一组你认为安全的协议里，这个命令像下面这样：

```
mysql> GRANT ... IDENTIFIED BY 'p4ssword'
-> REQUIRE CIPHER "EDH-RSA-DES-CBC3-SHA";
```

管理单独的客户端证书看上去是极好的安全方式，但是，它也会成为管理员的噩梦。当你创建了一个客户端证书后，你就必须给它设定一个失效日期——最好是在不远的将来。你希望它的寿命足够长，这样就不会使你被迫经常性地重新生成新的证书。如果寿命足够短，那即使是它落入了敌对方的手里，他们访问你的数据的时间也长不了。

在一个只有几名雇员的小环境里，持续追踪单个证书的所有权是件非常容易的事情。但是，当你的组织规模扩大到几百乃至几千名雇员的时候，要追踪哪个证书已经过期，并确保证书在过期前被更新掉，将会相当的繁琐。

一些组织解决此类问题的方法是结合 REQUIRE ISSUER 和一系列月度客户端证书——这些证书是通过可信任的发布渠道（例如企业内部局域网）来发布的。使用这些够用一个月或两个月的证书，客户端就能下载和连接到 MySQL 服务器。这样，如果一个雇员丢失了公司内部局域网的访问权，或者一个合作人不再拥有月度 key 的访问权，甚至是管理员没被告知要删除某个用户的访问权，那他的连接权限也会根据预订的时间表自然地失效。

SSH 隧道技术

如果你用的是一个老版本的 MySQL，或者仅仅不想被 SSL 的设置所烦扰，那就可以考虑用 SSH 来代替。如果正使用 Linux 或 Unix，那么其实你已经在用 SSH 登录到远程服务器上了。（注 8）许多人所不知道的是使用 SSH 还可以在两个主机之间建立一条加密的隧道。

SSH 隧道最好用一个例子来说明。让我们假定你想在一台 GNU/Linux 工作站和 *db.example.com* 上的 MySQL 服务器之间建立一个加密连接。在工作站这边，你需要执行下述命令：（注 9）

```
$ ssh -N -f -L 4406:db.example.com:3306
```

这在工作站的 TCP 4406 端口与 *db.example.com* 的 3306 端口之间建立了一个隧道。现在，你就能通过隧道从工作站连接到 MySQL 了，就像这样：

```
$ mysql -h 127.0.0.1 -P 4406
```

SSH 是一种非常强大的工具，它所能做的远超过这个简单示例所表明的。Stunnel 是另一种用于创建安全隧道的工具，但是，它没有登录/shell 组件。在一些情形下，它也是替代 VPN 的好工具。

注 8：有一种 OpenSSH 变体可用于 Windows 客户端，Putty (<http://www.chiark.greenend.org.uk/~sgtatham/putty/>) 的使用也比较广泛。<http://www.vbmysql.com/articles/security/sshtunnel.html> 是一个完整的教程，它能指导你如何建立一个 SSH 隧道去连接 MySQL 服务器。

注 9：假设 SSH v2 已经安装了。SSH v1 没有 -N 选项。更多细节可以查看 SSH 文档。

12.4.5 TCP 封装

你可以在 Unix 系统里把 TCP 封装的支持编译到 MySQL 里。如果没有使用一个成熟的防火墙，那么 TCP 封装也能提供最基本的防御功能：在不必更改你的授权表的情况下，你可以对那些 MySQL 要连接或不连接的主机施加更多一层的控制。有一些操作系统，例如 Debian GNU/Linux，在默认配置下就是这么构建 MySQL 的。

为了能使用 TCP 封装，你需要构建 MySQL 的源码，把 `--with-libwrap` 选项传给 `configure`，这样它就会知道在操作系统上的哪个地方才能找到合适的头文件：

```
$ ./configure --with-libwrap=/usr/local/tcp_wrappers
```

假设在你的 `/etc/hosts.deny` 文件里有一个条目，默认条件下是拒绝所有连接：

```
# deny all connections
ALL: ALL
```

你可以显式地把 MySQL 加入到你的 `/etc/hosts.allow` 文件里：

```
# allow mysql connections from hosts on the local network
mysqld: 192.168.1.0/255.255.0.0 : allow
```

唯一还要做的是为 MySQL 在 `/etc/services` 里增加一个合适的条目。如果你还没设置过，那就加入下面一行：

```
mysql      3306/tcp  # MySQL Server
```

如果你是在一个非标准端口上运行 MySQL——不是在 3306 端口上。

TCP 封装会增加一些系统开销，例如逆向 DNS 查找——这会在 DNS 子系统里创建一个依赖——这不是你所希望看到的。

12.4.6 自动主机封锁

Automatic Host Blocking

MySQL 在防止基于网络的攻击方面提供了一些帮助：如果它注意到有太多的有害连接来自一个特定的主机，它就会封锁所有来自该主机的连接。服务器变量 `max_connection_errors` 决定了 MySQL 在开始封锁主机前可以允许有多少个有害连接。“有害连接”指的是那些永不完成的连接（结果是一直占用着一个可用的 MySQL 会话）。拙劣的密码常常是有害连接的始作俑者，但是，网络问题也会导致有害连接的产生。

当 MySQL 封锁了一个主机时，它会把相应的消息写进日志，消息内容如下：

```
Host 'host.badguy.com' blocked because of many connection errors.
Unblock with 'mysqladmin flush-hosts'
```

如同消息显示的那样，在你弄明白了被封锁主机的连接问题，并采取了相应的措施之后，你可以使用 `mysqladmin flush-hosts` 命令让主机取消该封锁，`mysqladmin flush hosts` 命令仅仅执行了一个 `FLUSH HOSTS` 的 SQL 命令，它会清空 MySQL 的主机缓存表，还会解除所有被封锁主机的封锁，对于单独一个主机它无法做到。

出于某种原因，如果你发现有害连接的起因是一个普通问题，你就可以把 `my.cnf` 文件里的 `max_connection_errors` 变量设置为一个相对比较大的数值，以避免被封锁：

```
max_connection_errors=999999999
```

你不可能通过把 `max_connection_errors` 设为 0 来彻底关闭连接检查，而且通过任何其他办法你也无法做到。你最好还是找出并解决其背后的问题。

12.5 数据加密

Data Encryption

在应用里，常常存放着一些敏感数据，例如银行记录，你可能希望能以加密的形式来保存数据。对于那些未经授权的用户而言，使用这些加密了的数据将会很困难，哪怕他们能在物理上访问到你的服务器。全面讨论加密算法和技术会超出本书的内容范围，但是，我们会快速浏览一些跟它相关的主题。

12.5.1 密码散列

Password Encryption

在一些不怎么敏感的应用里，你需要保护的只有一些数量较少的信息，例如另外一个应用的密码数据库。密码不应该用明文来保存，因此，它们在应用里一般都是被加密的。但是，除了加密之外，更明智的做法是效仿大多数的 Unix 系统，甚至 MySQL 本身：对密码使用散列算法，把结果保存在你的表里。

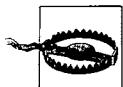
不同于传统的加密方法——它们都是可逆的，一个优秀的散列函数就是一个单向过程，是不可逆的。破解密码的唯一方法就是使用需要昂贵计算能力的暴力攻击生成一个个特定散列值进行匹配，穷尽所有可能的数值。

MySQL 提供了 3 个用户函数可用于密码的散列：`ENCRYPT()`、`SHA1()` 和 `MD5()`。（注 10）查看每个函数执行结果的最好方式就是把它们放在同一段源代码试运行，让我们看一下字符串 `p4ssword` 经过三个函数散列后的结果是怎么样的：

```
mysql> SELECT MD5('p4ssword'), ENCRYPT('p4ssword'), SHA1('p4ssword')\G
***** 1. row *****
MD5('p4ssword'): 93863810133ebebe6e4c6bbc2a6ce1e7
ENCRYPT('p4ssword'): dDCjeBzIycENk
SHA1('p4ssword'): fbb73ec5afd91d5b503ca11756e33d21a9045d9d
```

551

每一个函数都会返回一个固定长度的包含了字母和数字的字符串，你可以把它存储在一个 `CHAR` 列里。为了能应付使用 `ENCRYPT()` 而带来的混合了大小写字母的结果，这个列的类型最好还是 `CHAR BINARY`。



警告：千万不要在应用里使用 MySQL 内部的 `PASSWORD()` 函数，它的结果在各个版本的 MySQL 里各不相同。

存储散列数据非常简单，就像这样：

```
mysql> INSERT INTO user_table (user, pass) VALUES ('user', MD5('p4ssword'));
```

要验证一个密码，就运行一个 `SELECT` 查询，看看提供的用户名和密码是否匹配。若以 Perl 为例，你可以这样来做：

注 10：MySQL 的 `ENCRYPT()` 仅仅是调用了 C 语言库里的 `crypt()` 函数。在一些 Unix 里，`crypt()` 是一个 MD5 的实现，使它跟 `MD5()` 没什么区别。在其他操作系统里，它就是传统的 DES 加密算法。


```
my $sth = $dbh->prepare('SELECT * FROM user_table '
    . 'WHERE user = ? AND pass = MD5(?)');
$sth->execute($username, $password);
```

密码散列易于使用，而且相对比较安全，保存在数据库里也不容易被还原。再做一点改进就能使字典攻击更加困难：把用户名和密码放在一起做散列。这将使密码的产生要依赖更多的变量：

```
my $sth = $dbh->prepare('SELECT * FROM user_table '
    . 'WHERE user = ? AND pass = SHA1(CONCAT(?, ?))');
$sth->execute($username, $username, $password);
```

这里唯一还有安全风险的是用明文将密码发送给 MySQL，明文密码会被写入到磁盘上的日志文件里，并且在进程列表里是可见的。对此，你可以把密码作为用户变量存储起来，从而减少一点风险，或者把散列过程移到应用里（在大多数编程语言里，都有专门的加密函数和库），这样就彻底避免了泄密问题。在下文中，我们将会简短地浏览一下应用层的加密。

12.5.2 加密的文件系统

Database File System

MySQL 里的各种存储引擎都把它们的数据以普通文件的形式存储，不管你使用的是什么文件系统。所以，你也可以使用加密的文件系统。大多数流行的操作系统都至少有一种加密的文件系统可供使用（或者是免费的或者是商业的）。

这种方法的主要好处是你无须为了 MySQL 能利用加密文件系统的优点而特地去做些什么，因为所有加解密动作都是在 MySQL 之外完成的。MySQL 只要进行读写就可以，用不着任何关于底层操作细节的知识。所有你要做的事情就是确定把你的数据和日志文件都存放在正确的文件系统里。从你的应用那一端来看，是不是采用了这样的文件系统都一样，没什么区别的。

采用加密文件系统也有几个缺点。首先，因为你加密了所有数据、索引和日志，所以，在加解密这些信息的时候会产生大量的 CPU 开销。如果你正在考虑使用一个加密文件系统，那就一定要做个全面的基准测试，这样你就会了解你的服务器在重负载时的表现。

SQL

其次，要确保你做数据备份的时候，不会将这些数据都解密了。这不是个硬性规定，但是常常会忘记去做。

最后要知道的是一个加密文件系统对于那些能访问到数据所在的服务器的人来说，是没有任何保护的。因为文件系统的解密对于服务器来说是透明的，任何有访问服务器权限的人，也就能读到这些数据——甚至可以将这些数据做成一个解密了的副本。

12.5.3 应用层的加密

Application Level Encryption

一种更常用的方法是把加密功能构建到应用（或中间件）里。当应用需要存储敏感数据的时候，它首先会加密数据，然后再把结果存放到 MySQL 里。从相反的方向来说，当应用从 MySQL 那里接收一份加密数据后，它会先将它解密了。

这种方法提供了许多弹性。它不会把你跟特定的文件系统、操作系统、甚至是数据库（如果你的代码是以一种通用的方式编写的话）捆绑在一起。这也使应用设计者能够自由选择加密算法，让选中的加密算法可以更加适

合存储的数据（在加解密速度和加密强度之间平衡）。

因为数据是被加密的，所以备份非常容易。无论是从哪里复制数据，它都是被加密的。然而，这也意味着这些数据必须通过一个知道怎么解密的软件才能访问到。你再也不能打开一个命令行工具，然后发出查询要求了。

应用层加密一般都是个不错的解决方案，但是，它也有一些缺点。举例来说，MySQL 会难以高效地对加密数据做索引；当你在处理机密数据的时候，要想优化 MySQL 的性能也更加难了。

设计要点

我们在这里提到的自由和弹性跟数据设计还有着一些有趣的牵连。一个要点是你必须确保你使用的列的类型要跟加密类型相适合。有些算法产生的数据块有一个最小尺寸。这意味着你的列必须有 256 字节大小，刚好能容纳一块加密后的数据——它的尺寸小于加密前的尺寸。还有，许多流行的加密库都是产生二进制数据，所以，你创建的列也要能存储二进制数据。作为一个替代办法，你可以把二进制数据转换为一个十六进制或 base-64 来表示，但这会要求更多的空间和时间。

553 要决定哪些数据要加密哪些不要加密也不容易。你要在安全和表内信息查询的便利性之间做好权衡。举例来说，你有一个账号表，它代表了一系列的银行账号，每一行包含了下列这些字段：

- id
- type
- status
- balance
- overdraft_protection
- date_established

哪些列有加密的需要呢？如果你加密了“balance”，这看上去有点道理，但是会难以应付常用的查询问题。例如，你编写了下面这样一个查询，要在这些账号里找出最大、最小和平均的余额数：

```
mysql> SELECT MIN(balance), MAX(balance), AVG(balance)
-> FROM account GROUP BY type;
```

查询的结果将会毫无意义。MySQL 不知道这些加密了的 balance 列是何含义，因此，仅仅是针对这些加密数据执行了那几个函数。

对此的解决方案是从 account 表里读取所有的行，然后为你自己需要的报表做数学运算。这不是非常的难，只是有点令人讨厌，因为你不仅需要重新实现 MySQL 已提供的函数，而且还会减缓处理速度。

所有这些都可以归结为在安全与关系数据库优点之间的权衡。包含了加密数据的列对 MySQL 内嵌函数而言都是无用的，因为它们需要在未加密的数据上进行操作。同时，还有一个类似的问题，就是查询优化。举例来说，在一个未加密的数据表里，你可以很方便地找出所有余额大于\$100 000 的账号：

```
mysql> SELECT * FROM account WHERE balance > 100000;
```

如果在 balance 列上有一个未加密的索引，MySQL 就能利用这个索引来找到所需要的行。但是，如果数据是被加密了的，你就不得不把所有行都读到你的应用里，把它们解密后再进行过滤。

这就是说，你可以把加密后的值存放在 MySQL 里，并根据需要使用内嵌函数对它们进行加密、解密。在这方面用途里的最好的函数是 `AES_ENCRYPT()` 和 `AES_DECRYPT()`，它们能把一个字符串转换成加密的二进制字符串，或者反过来做。它们的加密算法是对称的：你加密时使用的 key 就是你解密时要用到的 key。例如：

```
mysql> SET @key      := 's3cret';
mysql> SET @encrypted := AES_ENCRYPT('sensitive data', @key);
mysql> SELECT AES_DECRYPT(@encrypted, @key);
+-----+
| AES_DECRYPT(@encrypted, @key) |
+-----+
| sensitive data                |
+-----+
```

我们没有显示出加密后的数值是因为它的二进制格式的结果显示出来就是一些乱码。

然而，这个方法也无法解决我们提及的所有问题。比如说它无法回避索引问题；还有你的加密数据在 SQL 查询语句里仍旧是明文形式，并被写入到系统日志里（假设日志功能是开启的）。但是，我们还是要给你指出一个步骤，可以降低别的用户看到你的加密数据的风险：把你的加密 key 存放在一个用户变量里，而保证用户变量安全性的方法就有很多了。例如，你可以把这个变量放在一个存储过程里，通过这个存储过程来设置它的值，同样，也能限制对这个存储过程的访问。这就能让其他用户难以确定这个 key 的值。

12.5.4 源代码修改

Source Code Modification

如果你想寻找一个比加密文件系统或基于应用的加密技术更具弹性的方法，那你可以自己来构建一个自定义的解决方案。MySQL 的源代码在 GNU General Public License 授权下可以自由使用。

这类工作要求你自己懂 C++，或者雇佣某个人来帮你做这个。满足了这个前提后，你就可以使用原生的加密支持来创建自己的存储引擎了，或者更简单一点，用加密支持扩展一个已有的存储引擎。

12.6 在 Chroot 环境里使用 MySQL

MySQL in a chrooted Environment

在 Unix 系统里，让服务器运行在 chroot 环境中能极大地提高整个系统的安全性。chroot 环境会建立一个独立的环境，指定目录以外的文件都无法被外界访问到。通过这种方法，即使服务器上有一个安全漏洞被发现并利用，其潜在的破坏也将被限定在那个目录的范围即那个特定的应用所使用的全部文件内。

如果想让你的 MySQL 应用运行在一个 chroot 环境里，你就必须这么开始：编译 MySQL 源代码，或者是解包并安装 MySQL AB 提供的二进制包。许多管理员认为这么做是理所当然的，但是这对于一个 chroot 化的应用来说却是必须的：许多预打包了的 MySQL 安装程序会把一些文件放在 `/usr/bin` 下，还有一些放在 `/var/lib/mysql` 下，等等；但是所有在一个 chroot 安装里的文件都要被放置在同一个目录结构下面。

我们将要做的是创建一个 `/chroot` 目录，所有 chroot 化的应用都会安装在那里。为了实现这个目的，你要像下面这样配置你的 MySQL 安装脚本：

```
$ ./configure --prefix=/chroot/mysql
```

然后像你通常做的一样编译 MySQL，让安装过程把 MySQL 文件都安装在 /chroot/mysql 下面。

接下来要做的就有一点奇妙了，它会让每一件事情更快乐。Chroot 实际上是 change roo 的缩写，如果你输入：

```
$ chroot /chroot/mysql
```

目录就是现在的 /chroot/mysql。因为 chroot 服务器和非 chroot 客户端都在使用这些文件，所以，建立一个统一的文件系统很重要，这样，无论服务器还是客户端都能找到它们所需要的文件。对于这个问题有个简单的解决办法，就像下面这样做：

```
$ cd /chroot/mysql
$ mkdir chroot
$ cd chroot
$ ln -s /chroot/mysql mysql
```

这会创建一个符号化的目录路径，/chroot/mysql/chroot/mysql 实际上指向的是 /chroot/mysql。哪怕应用在 chroot 之后，尝试着去访问 /chroot/mysql 时，它也能到达正确的目录。在此期间，如果一个客户端应用正在 chroot 环境之外运行，它也能找到它所需要的文件。

最后一个步骤就是发送特有的命令给 mysqld_safe，这样，MySQL 服务器就自己启动，并 chroot 到合适的目录下。为了实现这个步骤，你可以输入如下代码：

```
$ mysqld_safe --chroot=/chroot/mysql --user=1001
```

你会注意到我们使用了 MySQL 用户的 Unix UID(1001)来代替--user=mysql。这是因为在 chroot 环境下，MySQL 服务器不再能够查询到你的认证后端去做从用户名到 UID 的查找操作。（注 11）

这里还有一些使用 chrootMySQL 服务器时的告诫。LOAD DATA INFILE 及其他直接访问文件名的命令的行为方式会跟你所期望的有显著的不同，因为服务器不再把/当作文件系统的根目录。因此，如果你告诉它从 /tmp/filename 加载数据，你就要先确定文件是在 /chroot/mysql/tmp/filename 那里的，否则 MySQL 就无法找到它了。

Chroot 环境也不是部分隔离 MySQL 的唯一途径，还有其他的方法可供选用，例如 FreeBSD 的 jail、Solaris 的 Zone 和虚拟化技术。

556

注 11：在我们这个测试过程里，这跟在 chroot 环境里把 libnss* 复制到你的 MySQL 库目录里一样简单，但是，从实践的出发点来看，最好还是不要操心这种事情，应该是直接在启动脚本里输入 UID。

MySQL 服务器的状态

MySQL Server Status

通过查看 MySQL 的状态,你能回答很多关于 MySQL 服务器的问题。MySQL 主要通过两个途径发布它的内部信息:最新的方法是标准的 `INFORMATION_SCHEMA` 数据库,传统的方法是一系列的 `SHOW` 命令(这个方法 MySQL 还继续支持,即使已经有了更好的 `INFORMATION_SCHEMA` 数据库)。如果你在 `INFORMATION_SCHEMA` 表里有查不到的信息,那就可以通过 `SHOW` 命令来得到。

你面临的挑战是要决定哪些信息是跟你的问题相关的,如何得到所需要的信息,以及怎么解读它们。虽然 MySQL 让你看到了大量的服务器内部信息,但是,要利用这些信息也不总是很容易。理解这些信息需要耐心、经验和随时准备查阅 MySQL 使用手册。

有一些工具可以帮助你理解在不同上下文环境下的服务器状态,例如监控和分析。我们在下一章会讲到其中的一些。然而,你仍然需要从一个高层次上来理解这些值——最低限度要知道这些值是怎么分类的——还要知道怎么从服务器上读到它们。

本章要解释许多状态命令和它们的输出结果。如果我们提及的一个主题,在本书的其他地方曾作过详述的话,我们就会将你指引到那个部分去。

13.1 系统变量

System Variables

MySQL 通过 `SHOW VARIABLES` SQL 命令显露出许多系统变量。你可以在表达式里使用这些变量,或者在命令行里使用 `mysqladmin variables`。自从 MySQL 5.1 开始,你也可以通过 `INFORMATION_SCHEMA` 数据库里的各个表来访问这些变量了。

这些变量代表着多种配置信息,例如服务器默认的存储引擎 (`Storage_engine`)、当前的时区、连接的字符集和启动参数。我们在第 6 章里已经解释过如何设置和使用这些系统变量。

13.2 SHOW STATUS

`SHOW STATUS` 命令会在一个由两列(名称/值)组成的表格里显示服务器状态变量。跟上一节里我们提到的系统变量不同,这些状态变量都是只读的。你可以像执行 SQL 命令一样执行 `SHOW STATUS` 来显示这些变量;或者也可以像 shell 命令一样执行 `mysqladmin extended-status` 来显示。如果你使用的是 SQL 命令,那你也可以用上 `LIKE` 和 `WHERE` 来限制显示结果。`LIKE` 会对变量名做标准的模式匹配。命令执行后会返回一个表格形式的结果,但是,你无法对它进行排序、联接到别的表,也无法完成你原本能在 MySQL 表里做的一些标准操作。



提示：我们使用术语“状态变量”用来指代从 SHOW STATUS 读来的值；术语“系统变量”指代服务器配置变量。

SHOW STATUS 的行为在 MySQL 5.0 里变化比较大，但是，除非你仔细地注意过它，否则你是不会发现这一点的。原先 MySQL 保持着全局变量的一个集合，现在只保持了一部分全部变量和每个连接的一些基本变量。因而，SHOW STATUS 包含的是一个全局变量和会话变量的混合体，它们中的很多具备了两种作用范围：既是全局变量又是会话变量，并且有相同的名称。现在，SHOW STATUS 是默认地显示会话变量，因此，如果你已经习惯于用 SHOW STATUS 查看全局变量的话，你就再也看不到这些了，你只能使用 SHOW GLOBAL STATUS 来代替。（注 1）

在 MySQL 5.1 及更新的版本里，你能直接从 INFORMATION_SCHEMA.GLOBAL_STATUS 和 INFORMATION_SCHEMA.SESSION_STATUS 表里取出值来。

在 MySQL 5.0 服务器里就有好几百个状态变量，更新的版本里还会有更多。它们的大多数要么是计数值，要么是一些状态的当前值。计数器会在每次 MySQL 执行某个操作时将值累加上，例如初始化一个全表扫描（Select_scan）。度量值，例如服务器上打开着的连接数（Threads_connected），有时会增加有时减少。有时，某几个变量看上去会是指向同一个东西，例如 Connections（试图联接到服务器的连接数）和 Threads_connected。在这种情况下，这几个变量都是相互关联的，但是类似的名称并不总是意味着它们之间存在着某种联系。

计数器是以无符号整数类型存储的。在 32 位系统里，它占用了 4 个字节；在 64 位系统里是 8 个字节。当达到最大值之后，它们会归零。如果你正在监控那些值的增长，有需要注意它们的归零行为。如果你的服务器已经运行了很长一段时间，你就应该知道，某些值应该要比你所期望的小一些，因为它们已经归零过了。（这种问题在 64 位系统里很少会出现。）

559 查看这些变量的最好办法是在时长为几分钟的时间段里看它们改变了多少。你可以使用 mysqladmin extended-status -r-i 5 或 innotop 来做这个。

以下是对我们 SHOW STATUS 中看到的那些变量做一个大致的分类（不是完全的分类）。对于每个指定变量的细节，你要参考 MySQL 使用手册，<http://dev.mysql.com/doc/en/mysql-option-tables.html> 的页面内容就是关于这些变量的很有帮助的文档。在下面，当我们讨论一组具有同样前缀的相关变量时，会把它们叫做“<前缀 x>_变量”。

13.2.1 线程和连接的统计信息

Thread and Connection Statistics

这些变量追踪着尝试连接、退出连接、网络流量和线程的统计信息：

- Connections, Max_used_connections, Threads_connected
- Aborted_clients, Aborted_connects
- Bytes_received, Bytes_sent
- Slow_launch_threads, Threads_cached, Threads_created, Threads_running

注 1：这里有一个小把戏：如果你在新服务器上使用旧版本的 mysqladmin，是无法使用 SHOW GLOBAL STATUS 的，所以就无法显示出“正确”的信息。

如果 `Aborted_connects` 不为 0，这就意味着你有网络问题或者有人尝试连接并且失败了（也可能是因为他们使用了错误的密码或者连到了一个错误的数据库）。如果这个值太大了，那就会有严重的副作用：它会促使 MySQL 去屏蔽主机。更多细节请查看第 12 章。

`Aborted_clients` 跟上面这个变量有着相似的名称，却有着完全不同的含义。如果这个值在累加，那就意味着有应用错误在发生，例如程序员在退出程序时没有正确地关闭 MySQL 连接。但是，这个变量也并非总是昭示着大问题的存在。

有一个很有用的度量值是每秒钟创建的线程数目 (`Threads_created / Uptime`)。如果这个值跟 0 没差太多，这意味着你的线程缓冲区太小了，新来的连接在线程缓冲区里找不到空闲的线程可用。

对于这些变量和度量值，查看服务器最近几分钟里而不是整个正常运行的时间段的值是非常有用的。

13.2.2 二进制日志的状态

`SHOW BINARY LOGS`

`Binlog_cache_use` 和 `Binlog_cache_disk_use` 状态变量可以显示当前二进制日志缓存里保存了多少个事务，有多少个事务因为过于庞大而无法放入缓存，只能把它们的语句存储在一个临时文件里。我们已经在第 6 章里解释过如何改变二进制日志缓存的大小了。

13.2.3 命令计数器

`SHOW STATUS LIKE 'Com_*'`

`Com_*` 变量记录了已发出的每一种 SQL 或者 C API 命令的数目。举例来说，`Com_select` 记录的是 `SELECT` 语句的数目，`Com_change_db` 记录的是使用 `USE` 语句或者通过 C API 改变一条连接的默认数据库的次数。`Questions` 变量记录了服务器收到的查询和命令的总数。然而，因为有查询缓存的命中、关闭和退出连接、及其他可能因素的存在，所以，所有 `Com_*` 变量的总数并不完全相等。

`Com_admin_commands` 状态变量可能会非常大。它不仅记录了管理命令的数目，也记录了发给 MySQL 实例的 ping 请求的数目。这些请求是通过 C API 发出的，一般都是来自客户端代码，例如下面这样的 Perl 代码：

```
my $dbh = DBI->connect(...);
while ( $dbh && $dbh->ping ) {
    # Do something
}
```

这些 ping 请求都是“垃圾”请求。尽管它们不会给服务器增加多少负担，但毕竟也是浪费资源。我们在 ORM 系统里看到过：在每个查询之前都会 ping 服务器，这些都是毫无意义的；我们也在一些数据库抽象库里看到过：在每个查询前都要更改默认的数据库，这会产生数目巨大的 `Com_change_db` 命令。因此，最好是消除这两种做法。

13.2.4 临时文件和表

`SHOW VARIABLES LIKE 'tmp_*'`

你可以查看那些记录了 MySQL 创建临时表和文件次数的变量：


```
mysql> SHOW GLOBAL STATUS LIKE 'Created_tmp%';
```

13.2.5 Handler 操作

Handler Operations

Handler API 是 MySQL 与存储引擎之间的接口。Handler_* 变量记录了 Handler 的操作次数，例如 MySQL 要求存储引擎从一个索引里读取下一行数据的次数。研读你服务器里的 Handler_* 变量能让你看清楚服务器做得最多的是哪几种工作。Handler_* 对分析概要查询也很有用。你可以用下面这行命令查看到这些变量：

```
mysql> SHOW GLOBAL STATUS LIKE 'Handler_%';
```

13.2.6 MyISAM 索引键缓冲区

MyISAM Key Buffer

Key_* 变量包含了 MyISAM 索引键缓冲区的度量值和计数器。你可以用下面这行命令来查看这些变量：

```
mysql> SHOW GLOBAL STATUS LIKE 'Key_%';
```

在第 6 章里有关于怎样分析和调优索引键缓存的详细说明。

13.2.7 文件描述符

File Descriptors

如果你使用的主要是 MyISAM 存储引擎，那查看文件描述符的统计信息就显得尤为重要，因为它们能告诉你 MySQL 打开每一张表的 .frm、.MYI 和 .MYD 文件的频度分别是多少。InnoDB 把所有数据都存放在它的表空间文件里，因此，如果你主要使用的是 InnoDB，这些变量就不怎么重要了。你可以这样来查看 Open_* 变量：

```
mysql> SHOW GLOBAL STATUS LIKE 'Open_%';
```

第 6 章里说明了怎样对影响这些变量的设置项进行调优。

13.2.8 查询缓存

Query Cache

你可以通过 Qcache_* 状态变量来检查查询缓存。如果出于性能起见，你的查询都依赖于查询缓存，那么，这一组变量会是相当的重要。要检查它们，就使用：

```
mysql> SHOW GLOBAL STATUS LIKE 'Qcache_%';
```

关于怎样调优查询缓存的详细说明在本书的第 5 章。

13.2.9 各种类型的 SELECT

SELECT Types

Select_* 变量记录了各类型 SELECT 查询的次数。它们可以帮你查看到各种类型查询计划的比例关系。不幸的是，有一些类型的查询没有相应的状态变量，例如 UPDATE 和 REPLACE，然而，你可以通过查看 Handler_* 状态变量（刚刚讨论过的）来获知那些非 SELECT 查询的性能情况。要查看所有 Select_* 变量，可以这么做：

```
mysql> SHOW GLOBAL STATUS LIKE 'Select_%';
```

据我们判断，Select_* 状态变量可以按照开销的降序来排名：

Select_range

一种联接的数目，该联接在第一个表的索引的指定范围内做扫描。

Select_scan

对第一个表作全表扫描的连接数目。如果第一个表里的每一行都被加入了连接的话，它就不会出错。唯一不好的事情是如果你不想加入所有的行，那就没有索引能让你找到那些想要的记录。

Select_full_range_join

一种连接的数目，它使用表 n 里的一个值去获取表 $n+1$ 里某个引用索引范围内的行。根据不同的查询，它的开销有时会比 Select_scan 大，有时会小。

Select_range_check

一种连接的数目，它会为表 n 里的每一行，在表 $n+1$ 里对索引重新估值，看看哪个开销最廉价。这通常意味着表 $n+1$ 没有索引可用于这个连接。这种查询计划的开销非常高。

Select_full_join

显示一个交叉连接，或者一个在表里没有任何标准能匹配到的行的连接。检查过的行的总数就是每个表里行数的乘积。这经常是个很糟糕的事情。

在一个调优过的服务器里，最后两个变量的增长应该不怎么快。比较这两个计数器的值分别与正在执行的查询总数 (Com-select) 的比值，你有时会发现一个没优化好的工作负荷正在运行。如果其中有一个比值比总数高出几个百分点，这就说明你可能需要优化一下你的查询和/或数据配置了。

还有一个相关的状态变量叫 slow_queries，我们为缓慢查询日志开发的补丁可以帮助你查看某个查询是否需要全连接，是否使用了查询缓存等等。更多的内容请查看第 65 页的“更精细地控制日志”。

13.2.10 排序

Sorts

在第 3 章和第 4 章里，我们讲到了很多关于 MySQL 排序优化的内容，因此，对于排序的工作原理，你应该已经有了一个较好的认识。当 MySQL 无法使用索引去获得预排序的行时，它就不得不做文件排序，从而，会增加 Sort_* 状态变量的值。只能通过增加可用于排序的索引来影响这些值。Sort_merge_passes 依赖于服务器变量 sort_buffer_size（不要跟服务器变量 myisam_sort_buffer_size 搞混了。）MySQL 使用排序缓冲区存储一批数据行做排序。在完成排序之后，它会把这些已排序的行归并到结果里，并增加 Sort_merge_passes 的值，然后又把新的一批数据行填充到缓冲区里进行排序。如果排序缓冲区太小，这个过程会被重复很多次，于是这个状态变量的值将很大。

你可以这样来查看所有 Sort_* 变量：

```
mysql> SHOW GLOBAL STATUS LIKE 'Sort_%';
```

当 MySQL 从文件排序的结果里读出排序过的数据行，并返回给客户端时，它会增加 Sort_scan 和 Sort_range 两个变量的值。这两个变量的主要区别在于：前者是当查询方案触发了 Select_scan 增长时，它也会增长；后者是当 Select_range 增长时，它也随之增长。两者之间没有实现和开销方面的差异，主要是为了表明各自对应的查询方式触发排序的次数。

13.2.11 表锁定

Table Locking

Table_locks_immediate 和 Table_locks_waited 变量会告诉你这个时刻有多少个锁立即被授予，又有多少个需要等待。如果你在 SHOW FULL PROCESSLIST 看到有很多线程处于锁定状态，那你就检查一下这些值。然而，要知道显示出来的仅仅是服务器级锁的统计信息，没包含存储引擎的。关于如何调试锁，请查看附录 D。

13.2.12 Secure Sockets Layer (SSL)

Ssl_* 变量显示了服务器是如何配置 SSL 的（假设是正确地配置）。你可以这样查看所有 SSL 变量：

```
mysql> SHOW GLOBAL STATUS LIKE 'Ssl_%';
```

13.2.13 InnoDB 特有的变量

InnoDB-Specific

Innodb_* 变量显示的是包含在 SHOW INNODB STATUS（这个在本章的后半部分里会讲到）里的一些数据。这些变量可以通过名称来分组：Innodb_buffer_pool_*、Innodb_log_* 等。当我们检查 SHOW INNODB STATUS 时，会更多地讨论其内部的那些变量。

这些变量在 MySQL 5.0 及更新的版本里可以使用，它们有一个很重要的副作用：它们会创建一个全局锁，在释放锁之前，还会遍历整个 InnoDB 的缓存池，在此期间，其他线程都会阻塞，直到它把锁释放了。因此，它影响到了其他几个状态值，例如 Threads_running，它们会显示出高于平常的数值（可能还会高很多，具体要看你的服务器有多繁忙）。同样的影响也会在你运行 SHOW INNODB STATUS 或者通过 INFORMATION_SCHEMA 表（在 MySQL 5.0 及更新的版本里，SHOW STATUS 和 SHOW VARIABLES 在后台都被映射到 INFORMATION_SCHEMA 表的查询上）访问这些统计信息时发生。

因此，上面这些操作在这些版本的 MySQL 里会比较昂贵——过于频繁地查看服务器的状态（例如每秒钟一次）引发了数量可观的系统开销。使用 SHOW STATUS LIKE 也没用，因为它也是获取了全部的状态后再做过滤的。

13.2.14 Plug-in 特有的变量

Plug-in-Specific

MySQL 5.1 和更新的版本支持可插入的存储引擎，并为存储引擎提供了一个机制，用来注册它们自己的状态和配置变量。如果你正在使用一个插入式存储引擎，就会看到一个跟插入式有关的变量。

13.2.15 其他

Miscellaneous

我们将其他一些状态变量罗列在下面：

Delayed_*, Not_flushed_delayed_rows

这些变量是 INSERT DELAYED 查询的计数器和度量值。

564

Last_query_cost

这个变量显示了查询优化器的查询计划在最近一次执行查询时的开销。

我们在第 4 章里已经讨论过查询计划开销了。

Ndb_*

这个变量显示了 NDB Cluster 的配置信息（如果是正确的配置的话）。

Slave_*

当服务器是一个复制从服务器时，这个变量就启用了。Slave_open_temp_tables 变量对于基于语句的复制非常重要。更多关于复制和临时表的内容请查看第 394 页的“丢失的临时表”。

Tc_log_*

这个计数值用于记录该服务器被用作 XA 事务协调器的次数。更多细节请查看第 262 页的“分布式（XA）事务”。

Uptime

这个变量显示了服务器的正常运行时间，以秒为单位。

要想了解系统总体工作负荷的最好途径是比较一组相关的状态变量。例如，查看所有 Select_* 变量，或者 Handler_* 变量。如果你正在使用 innotop，采用 Command Summary 模式将会更方便做到这一点，但是，你也可以通过像 `mysqladmin extended -r -i60 | grep Handler_` 这样的命令来手动做这个事情。以下就是我们在某台服务器上使用 innotop 检查 Select_* 变量的结果显示：

| Command Summary | | | | | |
|------------------------|--------|--------|------|------|---------|
| Name | Value | Pct | Last | Incr | Pct |
| Select_scan | 756582 | 59.89% | | 2 | 100.00% |
| Select_range | 497675 | 39.40% | | 0 | 0.00% |
| Select_full_join | 7847 | 0.62% | | 0 | 0.00% |
| Select_full_range_join | 1159 | 0.09% | | 0 | 0.00% |
| Select_range_check | 1 | 0.00% | | 0 | 0.00% |

最前面两列的值是自服务器启动以来的累计数值；后面两列是自最近一次刷新以来的累计数值（在这个例子里，是 10 秒钟之前）。在显示的数据里，百分比超过了显示的数值总和，但是不会超过所有查询的总和。

即使是这台服务器有着百分比相对较低的全联接，它也值得仔细查看一番，查究为什么还会有一些全联接存在。

13.3 SHOW INNODB STATUS

在 SHOW ENGINE INNODB STATUS（或者一个相近的命令：SHOW INNODB STATUS）的输出里，InnoDB 存储引擎显示出了大量的内部信息。跟大多数 SHOW 命令不同的是，它的输出就是单独的字符串，没有行和列。它为很多个节，每一节对应了 InnoDB 存储引擎不同部分的信息，其中有一些信息对于 InnoDB 开发者来说是非常有用的，但是，许多信息如果你试着去理解，并且应用到高性能 InnoDB 调优的时候，你会发现它们都非常有趣——甚至是非常核心的。



提示：InnoDB 经常把 64 位数字分成两部分来打印：高 32 位和低 32 位。有一个例子就是事务 ID，比如 TRANSACTION 0 3793469，你可以这么来计算 64 位数字的值：把第一部分往左移动 32 位，然后把它加到第二部分上。我们在后面会展示几个例子。

输出内容包含了一些平均值的统计信息，例如 fsync() 在每一秒里的调用次数。这些显示的平均值是自上次输出结果生成以来的统计数，因此，如果你正在检查这些值，那就要确保你已经等待了 30 秒左右的时间，使两次采样之间积累起足够多的统计时间。不是所有的输出都会在一个时间点上就能生成，所以，也不是所有显示出

来的平均值会在同一时间间隔里被重新计算一遍。而且，InnoDB 有一个内部复位间隔，它是不可预知的，在各个版本里也是不一样的。您应该检查一下输出，看看有哪些平均值在这个时间段里生成，因为两次采样的时间间隔不必总是相同的。

这里面有足够的信息让你手工计算出大多数你想要的统计信息。但是，如果这时有一款监控工具，例如 innotop——它能为你计算出增量差别和平均值——那将是非常有用的。

13.3.1 头部信息

输出的第 1 段是头部信息，它仅仅是代表输出的开始，其内容包括当前的日期和时间，以及自上次输出以来经过的时长。第 2 行是当前日期和时间；第 4 行显示的是平均值计算的时间间隔，它或者是上次输出以来的时间，或者是距离上次内部复位的时长：

```
566 1 =====
2 070913 10:31:48 INNODB MONITOR OUTPUT
3 =====
4 Per second averages calculated from the last 49 seconds
```

13.3.2 SEMAPHORE

如果你有高并发的的工作负荷，你就要关注这一段信号量。它包含了两种数据：时间计数器，以及可选的当前等待线程的列表。如果有性能上的瓶颈，你就可以用这些信息来帮你找出瓶颈到底在哪里。不幸的是，要想知道怎么使用这些信息还是有一点点复杂，但是，我们会在本章的后半部里给你一些建议。这里是一些输出样例：

```
1 -----
2 SEMAPHORES
3 -----
4 OS WAIT ARRAY INFO: reservation count 13569, signal count 11421
5 --Thread 1152170336 has waited at ../../include/buf0buf.ic line 630 for 0.00 seconds
  the semaphore:
6 Mutex at 0x2a957858b8 created file buf0buf.c line 517, lock var 0
7 waiters flag 0
8 wait is ending
9 --Thread 1147709792 has waited at ../../include/buf0buf.ic line 630 for 0.00 seconds
  the semaphore:
10 Mutex at 0x2a957858b8 created file buf0buf.c line 517, lock var 0
11 waiters flag 0
12 wait is ending
13 Mutex spin waits 5672442, rounds 3899888, OS waits 4719
14 RW-shared spins 5920, OS waits 2918; RW-excl spins 3463, OS waits 3163
```

第 4 行给出了关于操作系统等待阵列的信息，它是一个“插槽”式的阵列，InnoDB 保留了阵列里的一些插槽给信号量使用，操作系统用这些信号量给线程发送信号，使线程可以继续运行，完成它们等着要做的事情。这一行还显示出 InnoDB 使用了多少次操作系统的等待。reservation count 显示了 InnoDB 分配插槽的频度，而 signal count 衡量的是线程通过阵列得到信号的频度。操作系统的等待相对于循环等待（Spin wait）要更昂贵一些，我们即将看到这一点。

第 5 到 12 行显示的是当前正在等待互斥量的 InnoDB 线程。在这个例子里显示出有两个线程正在等待，每一个都是以“-- Thread <num> has waited...”开始的。这一段应该是空的，除非你的服务器运行着高并发的的工作负载，

它促使 InnoDB 采取让操作系统等待的措施。如果你对 InnoDB 源代码很熟悉的话，你在这里会看到非常有用的信息：发生线程等待的代码文件名。这就给了你一个提示：在 InnoDB 内部哪里才是热点。举例来说，你看到许多线程都在一个名为 `buf0buf.ic` 的文件上等待着，这意味着你的系统里存在着缓冲池竞争。这个输入信息还显示了这些线程等待了多长的时间，另外，“waiters flag”显示了有多少个等待者正在等待同一个互斥量，文字“wait is ending”意味着这个互斥量实际上已经被释放了，但是，操作系统还没把线程调度过来运行。

你可能弄不明白 InnoDB 真正等待的是什麼。InnoDB 使用了互斥量和信号量来保护代码的临界区，例如限定每次只能有一个线程进入临界区，或者是当有活动的读者时，就限制写者的加入等等。在 InnoDB 代码里有很多临界区，在正确的条件下，它们都会显示在那里。你常常能见到的一个就是获取缓冲池分页的访问权。

在等待线程的列表之后，第 13、14 行显示了更多的事件计数器。第 13 行显示的是跟互斥量相关的几个计数器，第 14 行用于显示读/写共享和排斥锁的计数器。在每一个案例里，你都能看到 InnoDB 诉诸于操作系统等待的频率。

InnoDB 有着一个多阶段等待策略。首先，它会试着对锁进行循环等待。如果经过了一个预设的循环等待周期（设置 `innodb_sync_spin_loops` 配置变量）之后还没有成功，那就会退到更昂贵更复杂的等待阵列里（注 2）。

循环等待的成本相对较低，但是它们要不停地检查一个资源是否被锁定，还是消耗了 CPU 周期。但是，这没有像它听起来那么糟糕，因为当处理器在等待 I/O 时，一般都有一些空闲的 CPU 周期可用，即使是没有空闲的 CPU 周期，空等也要比其他方式更加廉价一些。然而，当另外一条线程能做一些事情时，循环等待也会独占处理器。

循环等待的替换方案就是让操作系统做上下文切换，这样，当这条线程在等待时，另外一条线程就可以被运行，然后，通过等待阵列里的信号量发出信号，唤醒那条沉睡的线程。通过信号量来发送信号是比较有效率的，但是，上下文切换就很昂贵。你很快就能累加出来：每秒钟几千次的切换会引发大量的系统开销。

你可以通过改变系统变量 `innodb_sync_spin_loops` 的值，试着在循环等待与操作系统等待之间作一个平衡。不要担心循环等待，除非你在每一秒里看到了许多循环等待（大概是成百上千个）。在第 6 章里，有更多关于如何调优这些变量的建议。

13.3.3 LATEST FOREIGN KEY ERROR

在下一段里，LATEST FOREIGN KEY ERROR 一般不会出现，除非你的服务器上有一个外键错误。在源代码里有许多地方会生成这样的输出，具体还跟错误的类型有关系，有时是一个事务在插入、更新或删除一条记录时要寻找到父行或子行；或者是在改变一个定义了外键的表结构时。

这一段的输出对于调试 InnoDB 常常发生的原因模糊的外键错误非常有帮助。让我们来看几个实例。首先，我们创建两个表，并在两者之间建立一个外键，并插入一些数据：

```
CREATE TABLE parent (
  parent_id int NOT NULL,
  PRIMARY KEY(parent_id)
) ENGINE=InnoDB;

CREATE TABLE child (
  parent_id int NOT NULL,
```

注 2：等待阵列在 MySQL 5.1 里变得更加高效了。


```

    KEY parent_id (parent_id),
    CONSTRAINT child_ibfk_1 FOREIGN KEY (parent_id) REFERENCES parent (parent_id)
) ENGINE=InnoDB;

INSERT INTO parent(parent_id) VALUES(1);
INSERT INTO child(parent_id) VALUES(1);

```

这里有两类基本的外键错误。因为增加、更新或删除数据而违反外键约束的是第一类错误。举例来说，以下就是当我们从父表里删除一行数据时发生的错误：

```

DELETE FROM parent;
ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key constraint
fails (`test/child`, CONSTRAINT `child_ibfk_1` FOREIGN KEY (`parent_id`) REFERENCES
`parent` (`parent_id`))

```

这个错误信息非常直观易懂。当你增加、更新和删除没有匹配的行时，都会看到跟它相似的错误信息。下面是通过 SHOW INNODB STATUS 看到的结果：

```

1  -----
2  LATEST FOREIGN KEY ERROR
3  -----
4  070913 10:57:34 Transaction:
5  TRANSACTION 0 3793469, ACTIVE 0 sec, process no 5488, OS thread id 1141152064 updating
   or deleting, thread declared inside InnoDB 499
6  mysql tables in use 1, locked 1
7  4 lock struct(s), heap size 1216, undo log entries 1
8  MySQL thread id 9, query id 305 localhost baron updating
9  DELETE FROM parent
10 Foreign key constraint fails for table `test/child`:
11 ,
12  CONSTRAINT `child_ibfk_1` FOREIGN KEY (`parent_id`) REFERENCES `parent`
   (`parent_id`)
13 Trying to delete or update in parent table, in index `PRIMARY` tuple:
14 DATA TUPLE: 3 fields;
15  0: len 4; hex 80000001; asc    ;; 1: len 6; hex 00000039e23d; asc    9 =;; 2: len 7;
   hex 000000002d0e24; asc    - $;;
16
17 But in child table `test/child`, in index `parent_id`, there is a record:
18 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
19  0: len 4; hex 80000001; asc    ;; 1: len 6; hex 000000000500; asc    ;;

```

第 4 行显示了最近的那个外键错误发生的日期和时间。第 5 到第 9 行显示的是那个引发外键错误的事务的细节，我们会在后面解释这一行。第 10 到 19 行显示的是当错误发生时，InnoDB 正在更改的数据。这些输入的大部分是转换为可打印格式的数据行，对此，我们也会在后面做一些说明。

直到现在，一切都还好理解。但是，另外一种外键错误就非常难以调试了。以下就是当我们试着更改父表结构时发生的错误：

```

ALTER TABLE parent MODIFY parent_id INT UNSIGNED NOT NULL;
ERROR 1025 (HY000): Error on rename of './test/#sql-1570_9' to './test/parent'
(errno: 150)

```

这看上去不够清晰，但是，用 SHOW INNODB STATUS 来查看会更明显一些：

```

1  -----
2  LATEST FOREIGN KEY ERROR
3  -----
4  070913 11:06:03 Error in foreign key constraint of table test/child:
5  there is no index in referenced table which would contain

```



```

6 the columns as the first columns, or the data types in the
7 referenced table do not match to the ones in table. Constraint:
8 ,
9 CONSTRAINT child_ibfk_1 FOREIGN KEY (parent_id) REFERENCES parent (parent_id)
10 The index in the foreign key in table is parent_id
11 See http://dev.mysql.com/doc/refman/5.0/en/innodb-foreign-key-constraints.html
12 for correct foreign key definition.

```

这里的错误原因源自不同的数据类型。作为外键的列必须有同样的数据类型，包括任何修饰符的更改（例如在这里是 UNSIGNED 引发的问题）。无论什么时候当你看到 1 025 号错误，不明白它为什么出现时，最好的办法就是查看 SHOW INNODB STATUS 的输出。

13.3.4 LATEST DETECTED DEADLOCK

跟上面的外键节一样，LATEST DETECTED DEADLOCK 也只有当服务器内有死锁时才会出现。

死锁在等待关系图（Waits-for Graph）里就是一个循环，就是一个锁定了行的数据结构又在等待别的锁。这个循环可以任意地大。InnoDB 会立即检测到死锁，因为每当有事务等待行锁的时候，它都会去检查等待关系图里是否有循环。死锁的情况可能会比较复杂，但是，这一段里只显示了最近两个死锁的情况，它们在各自事务里执行的最后一条语句，以及它们在图里形成循环的锁的信息。你看不到在这个循环里的其他事务，也看不到在事务里早先真正获得了锁的语句。尽管如此，你通常还是可以通过查看输入结果来确定到底是什么引起了死锁。

在 InnoDB 里实际上有两种死锁。第一种就是人们常常碰到的那种，它在等待关系图里是一个真正的循环。另外一种就是在一个等待关系图里，因代价昂贵而无法检查它是不是包含了循环。如果 InnoDB 要在图里检查超过 100 万个锁，或者在检查过程中，InnoDB 要重做 200 个以上的事务，那它就会放弃，并宣布这里有一个死锁。这些数值都是硬编码在 InnoDB 代码里的常量，你无法配置它们（如果你愿意的话，可以在代码里更改这些数值，然后重新编译）。当 InnoDB 的检查工作超过这个极限后，它就会引发一个死锁，这时你就在输出里看到一条信息 “TOO DEEP OR LONG SEARCH IN THE LOCK TABLE WAITS-FOR GRAPH”。

InnoDB 不仅会打印出事务和事务持有的锁和等待的锁，而且还有记录本身。这些信息对于 InnoDB 开发者来说都是很有用的，但是目前也没有办法关闭它。不幸的是，它会变得很大，超过你为输出结果预留的长度，以至于你都没法看到下面几段输出信息了。对此唯一的补救办法是，制造一个小死锁来替换那个大的死锁，或者使用由本书一个作者提供的补丁，它在 <http://lists.mysql.com/internals/35174> 可以得到。

这里有一个死锁信息的样例：

```

1 -----
2 LATEST DETECTED DEADLOCK
3 -----
4 070913 11:14:21
5 *** (1) TRANSACTION:
6 TRANSACTION 0 3793488, ACTIVE 2 sec, process no 5488, OS thread id 1141287232 starting
  index read
7 mysql tables in use 1, locked 1
8 LOCK WAIT 4 lock struct(s), heap size 1216
9 MySQL thread id 11, query id 350 localhost baron Updating
10 UPDATE test.tiny_d1 SET a = 0 WHERE a <> 0
11 *** (1) WAITING FOR THIS LOCK TO BE GRANTED:
12 RECORD LOCKS space id 0 page no 3662 n bits 72 index 'GEN_CLUST_INDEX' of table
  'test/tiny_d1' trx id 0 3793488 lock_mode X waiting
13 Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
14 0: len 6; hex 000000000501 ...[ omitted ] ...

```

```

15
16 *** (2) TRANSACTION:
17 TRANSACTION 0 3793489, ACTIVE 2 sec, process no 5488, OS thread id 1141422400 starting
   index read, thread declared inside InnoDB 500
18 mysql tables in use 1, locked 1
19 4 lock struct(s), heap size 1216
20 MySQL thread id 12, query id 351 localhost baron Updating
21 UPDATE test.tiny_d1 SET a = 1 WHERE a <> 1
22 *** (2) HOLDS THE LOCK(S):
23 RECORD LOCKS space id 0 page no 3662 n bits 72 index 'GEN_CLUST_INDEX' of table
   'test/tiny_d1' trx id 0 3793489 lock mode S
24 Record lock, heap no 1 PHYSICAL RECORD: n_fields 1; compact format; info bits 0
25 0: ... [ omitted ] ...
26
27 *** (2) WAITING FOR THIS LOCK TO BE GRANTED:
28 RECORD LOCKS space id 0 page no 3662 n bits 72 index 'GEN_CLUST_INDEX' of table
   'test/tiny_d1' trx id 0 3793489 lock mode X waiting
29 Record lock, heap no 2 PHYSICAL RECORD: n_fields 4; compact format; info bits 0
30 0: len 6; hex 000000000501 ...[ omitted ] ...
31
32 *** WE ROLL BACK TRANSACTION (2)

```

第 4 行显示的是死锁发生的时间，第 5 到 10 行信息的是死锁里的第一个事务的信息。在下一节里，我们会详尽地解释这些输出的含义。

第 11 到 15 行显示的是当死锁发生时，事务 1 正在等待的锁。我们忽略了其中第 14 行的信息，那是因为这只对调试才有用。这里要特别注意的内容是第 12 行，它告诉你这个事务正在等待的是 `test.tiny_d1` 表里的 `GEN_CLUST_INDEX`（注 3）的排它锁。

第 16 到 21 行显示的是第二个事务的状态，第 22 到 26 行显示的是该事务持有的锁。为了简洁起见，第 25 行的几条记录已经被我们删去了。这些记录里的一条就是第一个事务正在等待的是哪一条记录。最后，第 27 到 31 行显示它正在等待的是哪一个锁。

当一个事务持有了其他事务需要的锁，同时，它又想获取其他事务持有的锁，这时，等待关系图上的循环就产生了。InnoDB 不会显示所有持有的锁和等待的锁，但是，它显示了足够的信息帮你决定：查询操作正在使用的是哪些索引？这对于你是否能避免死锁有着极大的价值。

如果你能使两个查询在同一个索引、同一个方向上进行扫描，你就能降低死锁的数目，因为，查询在同一个顺序上请求锁的时候不会创建循环。有时候，这是很容易做到的，举例来说，如果你要在一个事务里更新许多条记录，就可以在应用程序的内存里把它们按主键进行排序，然后，再用同样的顺序更新到数据库——这样就不会有死锁的发生。但是，在其他一些时候，这个方法也是行不通的（例如当你有两个进程使用了不同的索引区操作同一张表的时候）。

第 32 行显示的是哪个事务被选中成为死锁的牺牲品。InnoDB 会把看上去最容易回滚（就是更新的记录数最少的）的事务选为牺牲品。

这些信息对于监控和日志分析是很有价值的。使用 Maatkit 的 `mk-dead-lock-logger` 工具来做这个非常方便，对于普通的日志，它也能派得上用场：找出所有与死锁相关的线程在执行的查询语句，看看到底是哪条语句引起了死锁。在下一节里，你会看到如何在死锁的输出信息里找到线程 ID。

注 3：这个索引是当你没有指定主键时，InnoDB 在内部自动创建的。

这一段里包含的是 InnoDB 事务的一些摘要信息，它们跟随在当前活动事务目录之后。以下就是前几行（头部信息）：

```

1  -----
2  TRANSACTIONS
3  -----
4  Trx id counter 0 80157601
5  Purge done for trx's n:o <0 80154573 undo n:o <0 0
6  History list length 6
7  Total number of lock structs in row lock hash table 0

```

具体的输出信息会根据 MySQL 版本的不同而有变化，但是至少包含了如下这些信息：

- 第 4 行：当前事务的 ID，它是一个系统变量，每创建一个新事务就会累加。
- 第 5 行：这是 InnoDB 清除旧版本 MVCC 行时所用的事务 ID。通过这个值和当前事务 ID 的比较，你会看到有多少老版本的数据已经被清除了。在这里，读取这个数字需要准备多大的取值范围才够安全没有硬性规定。如果数据没做过任何更新，一个巨大的数字也不意味着这是未清除的数据，因为实际上所有事务在数据库里查看的都是同一个版本的数据。从另一方面来讲，如果同时有很多行被更新，那每一行就会有一个或多个版本会被留在内存里。减少此类开销的最好办法是确认当事务完成时就立即将它提交，不要让它长时间地处于打开的状态。因为即使只有一个打开的事务不做任何操作，它也会影响到 InnoDB 清除旧版本的行数据。

同样是在第 5 行里，还有一项 InnoDB 清除进程正在使用的撤销日志编号——如果有的话。如在本例当中一样，如果它是“00”，说明清除进程处于空闲状态。

- 第 6 行：历史记录的长度，它就是位于 InnoDB 数据文件的撤销空间里的未清除事务的数目。当一个事务执行了更新并提交后，这个数字就会累加；当清除进程移除一个旧版本数据时，它就会递减。在第 5 行里，清除进程也会更新该数值。
- 第 7 行：锁结构的数目。每一个锁结构经常持有许多个行锁，所以，它跟被锁定行的数目不一样。

头部信息之后就是一个事务列表。当前版本的 MySQL 还不支持嵌套式事务，因此，在某个时间点上，每个客户端连接能拥有的事务数目是有一个上限的，而且每一个事务只能属于单独一个连接。在输出信息里，每一个事务至少占有两行内容。下面这个例子就是你能看到的一个事务所显示的最少的信息：

```

1  ---TRANSACTION 0 3793494, not started, process no 5488, OS thread id 1141152064
2  MySQL thread id 15, query id 479 localhost baron

```

第 1 行是以事务 ID 和状态开始的。这个事务正处于“not started”状态，这意味着它已经被提交，但是没有发出会影响到其他事务的执行语句，它可能就是空闲着。接下来就是一些进程和线程的信息。第 2 行显示的是 MySQL 的进程 ID，它跟 SHOW FULL PROCESSLIST 里显示的 Id 列是一致的。紧跟它之后的是一个内部查询号码和一些连接信息（这也跟你在 SHOW FULL PROCESSLIST 里发现的一样）。

其实，每一个事务都能打印出比上面更多的信息，这里就是一个更复杂的例子：

```

1  ---TRANSACTION 0 80157600, ACTIVE 4 sec, process no 3396, OS thread id 1148250464, thread
   declared inside InnoDB 442
2  mysql tables in use 1, locked 0

```

```

3 MySQL thread id 8079, query id 728899 localhost baron Sending data
4 select sql_calc_found_rows * from b limit 5
5 Trx read view will not see trx with id>= 0 80157601, sees <0 80157597

```

本例中的第 1 行显示出这个事务活动了 4 秒钟。它可能的状态是“not started”、“active”、“prepared”和“committed in memory”（一旦被提交到磁盘上，状态就会变为“not started”）。虽然在这个示例里没显示出来，但是，在其他条件下，你也能看到关于事务当前正在做什么的信息，有超过 30 个字符串常量可以作为这类信息显示在这里，例如“fetching rows”、“adding foreign keys”等等。

第 1 行里的“thread declared inside InnoDB 442”的意思是该线程正在 InnoDB 内核里做一些操作，并且，还有 442 张“票子”可以使用。换句话说就是同样的 SQL 查询可以重入 InnoDB 内核 442 次。这“票子”是系统用来限制内核中线程并发操作的手段，防止它在某些平台上运行失常。即使线程的状态是“inside InnoDB”，那线程也可能不必把所有的工作都放在 InnoDB 里面来做。查询大概也就是在服务器级上做一些操作，然后只要通过某个途径跟 InnoDB 内核互动一下就可以了。你会看到这些事务的状态是“sleeping before joining InnoDB queue”或者“waiting in InnoDB queue”。

接下来你看到的这一行显示了当前语句里使用到的和锁定的表有多少。InnoDB 不是以通常方式锁住表，有些语句还是可以在上面执行。当 MySQL 服务器在高于 InnoDB 层次之上将表锁定时，这里也是能够显示出来的。如果事务已经锁定了任意几行数据，这里将会有一行信息显示锁定结构的数目（再声明一次，这跟行锁是两回事）和堆的大小。具体例子你可以查看上文中死锁的输出信息。在 MySQL 5.1 及更新的版本里，这一行也是显示当前事务持有的行锁的实际数目。

堆的大小指的是为了持有这些行锁而占用的内存大小。InnoDB 是用一种特殊的位图表来实现行锁的，从理论上讲，它将每一个锁定的行表示为一个比特。我们的测试显示，每一个锁通常不超过 4 比特。

574 本例中的第 3 行包含的信息略微多于上一个样例中的第 2 行：在该行的末尾是线程状态“Sending data”，这跟你在 SHOW FULL PROCESSLIST 的 Command 列上看到的一样。

如果事务正在运行的一个查询，在本例的第 4 行里就会显示出查询的文本（或者，在某些版本的 MySQL 里，显示的是查询的摘要信息）。

第 5 行显示了事务的读视图，它表明了事务 ID 的范围，包括了因为版本关系而产生的对于事务可见的和不可见两种类型。在本例中，四个事务在两个数字之间有一个缺口，这四个事务可以是可见的，也可以是不可见的。当 InnoDB 执行一个查询时，对于那些事务 ID 正好落入这个缺口的行来说，InnoDB 还会检查它们的可见性。

如果事务正在等待一个锁，你就会在查询内容之后看到这个锁的信息。在上文的死锁例子里，这样的信息已经看到过多次了。不幸的是，输出信息并没有说出这个锁正被其他哪个事务持有着。

如果输出信息里有很多个事务，InnoDB 就限制了要打印出来的事务数目，以免输出信息变得很长。这时，你就会看到“...truncated...”字样。

13.3.6 FILE I/O

FILE I/O 段显示的是 I/O helper 线程的状态，用性能计数器的方式来表示：

```

1 -----
2 FILE I/O
3 -----
4 I/O thread 0 state: waiting for i/o request (insert buffer thread)

```

```

5 I/O thread 1 state: waiting for i/o request (log thread)
6 I/O thread 2 state: waiting for i/o request (read thread)
7 I/O thread 3 state: waiting for i/o request (write thread)
8 Pending normal aio reads: 0, aio writes: 0,
9   ibuf aio reads: 0, log i/o's: 0, sync i/o's: 0
10 Pending flushes (fsync) log: 0; buffer pool: 0
11 17909940 OS file reads, 22088963 OS file writes, 1743764 OS fsyncs
12 0.20 reads/s, 16384 avg bytes/read, 5.00 writes/s, 0.80 fsyncs/s

```

第 4 到 7 行显示的就是 I/O helper 线程当前状态。第 8 到 10 行显示的是每个 helper 线程还没完成的操作的数目，以及日志和缓冲池线程还没完成的 `fsync()` 操作的数目。其中的缩写词“aio”的意思是“asynchronous I/O”。第 11 行显示了读、写及 `fsync()` 调用的执行次数。这些变量适合用一个图形化的有趋势显示功能的系统来监控，我们会在下一章里谈到这样的系统。变量的绝对值会因工作负载的不同而有变化，因此，监控它们的变化情况显得尤为重要。第 12 行显示的是在头部信息指明的那段时间里，平均每一秒钟执行各种操作的次数。

第 8 到 9 行里显示的未决数值是检测 I/O 密集应用的好办法。如果这些类型的 IO 里大多数都有一些未决的操作，这说明当前的工作负载可能就是 I/O 密集的。

在 Windows 上，你可以通过 `innodb_file_io_threads` 配置变量来调整 I/O helper 线程的数目，这样，你就可以看到超过一条的读和写线程了。但是，在所有平台上，你至少会看到以下四条线程：

插入缓冲线程

负责插入缓冲区的合并（例如将记录从插入缓冲区合并到表空间里）。

日志线程

负责异步的日志刷新。

读线程

负责读前置（Read-ahead）的操作，预测 InnoDB 将要使用的数据，并将它们预读取进来。

写线程

刷新“脏”缓冲区。

13.3.7 INSERT BUFFER AND ADAPTIVE HASH INDEX

这一节显示的是 INSERT BUFFER AND ADAPTIVE HASH INDEX 的状态：

```

1 -----
2 INSERT BUFFER AND ADAPTIVE HASH INDEX
3 -----
4 Ibuf for space 0: size 1, free list len 887, seg size 889, is not empty
5 Ibuf for space 1: size 1, free list len 887, seg size 889,
6 2431891 inserts, 2672643 merged recs, 1059730 merges
7 Hash table size 8850487, used cells 2381348, node heap has 4091 buffer(s)
8 2208.17 hash searches/s, 175.05 non-hash searches/s

```

第 4 行显示的信息是插入缓冲区的大小、它的“自由列表”的长度、以及它的分段大小。其中的文本“for space 0”像是说明了多缓冲区（每个表空间一个）的可能性——但是这从没被实现过，而且这行文本在最近的 MySQL 版本里已经被移除了。插入缓冲区只可能有一个，因此，第 5 行是真正的冗余。第 6 行显示了 InnoDB 已经做了多少次缓冲区操作。查看其中合并到插入缓冲区的比例是评判缓冲区效率的好办法。

第 7 行显示的是自适应散列索引(Adaptive Hash Index)的状态。第 8 行显示了根据头部信息划定的期限内 InnoDB 做过的散列索引的次数。散列索引查找次数和非散列索引查找次数的比例是另一项效率度量值, 因为散列查找快过非散列查找。这些都是咨询式的信息, 你无法配置自适应散列索引。

576 13.3.8 LOG

这一段里显示的 InnoDB 事务 LOG 子系统的统计信息:

```
1 ---
2 LOG
3 ---
4 Log sequence number 84 3000620880
5 Log flushed up to 84 3000611265
6 Last checkpoint at 84 2939889199
7 0 pending log writes, 0 pending chkp writes
8 14073669 log i/o's done, 10.90 log i/o's/second
```

第 4 行显示了当前日志的顺序号, 第 5 行显示了日志已经刷新到的点位。日志顺序号就是已经写入日志文件的字节总数, 因此, 你能用它来计算日志缓冲区里还有多少数据没被刷新到日志文件里。在本例中, 是 9 615 个字节 (13000620880 - 13000611265)。第 6 行显示了最近的一个检查点 (一个检查点代表了一个瞬间, 在那一刻里数据和日志文件都处于可知的状态, 它可被用来做还原)。第 7 到 8 行显示了还未完成的日志操作和统计信息, 拿这些跟 FILE I/O 段的信息比较, 你会发现 I/O 里有多少是跟日志子系统相关的操作触发的。

13.3.9 BUFFER POOL AND MEMORY

这一节里显示的是 InnoDB 的 BUFFER POOL AND MEMORY 的统计信息。(关于如何调优缓冲池的信息, 请查看第 6 章。)

```
1 -----
2 BUFFER POOL AND MEMORY
3 -----
4 Total memory allocated 4648979546; in additional pool allocated 16773888
5 Buffer pool size 262144
6 Free buffers 0
7 Database pages 258053
8 Modified db pages 37491
9 Pending reads 0
10 Pending writes: LRU 0, flush list 0, single page 0
11 Pages read 57973114, created 251137, written 10761167
12 9.79 reads/s, 0.31 creates/s, 6.00 writes/s
13 Buffer pool hit rate 999 / 1000
```

第 4 行显示了 InnoDB 申请的内存总数, 以及其中有多少是在附加内存池里申请到的。

第 5 到 8 行显示了缓冲池的度量值, 以页为单位。这些度量值包括了缓冲池的总大小、空闲页的数量、用于存储数据库页的数量、和“脏”数据库页的数量。InnoDB 在缓冲池里把一些页面用作锁的索引、自适应索引散列及其他系统结果, 因此, 在池里的数据库页的数量永远不会等于池的总大小。

577 第 9 到 10 行显示了未完成的读和写的数目 (例如 InnoDB 要为缓冲池做的逻辑读和写的数目)。这些值会跟 FILE I/O 段里的数据不匹配, 因为 InnoDB 可能会将许多逻辑操作合并为一个单独的物理 I/O 操作。缩写语 LRU 代表了“最少最近使用”, 它在缓冲池里把不常用的页刷新到次磁盘上, 从而为常用的页腾出空间。刷新列表里保

存的一些旧页，它们须由检查点进程来刷新，单页写属于独立的页写操作，它们不会被合并掉。

第 8 行显示了这个缓冲池里容纳了 37 491 个脏页，在某个点上需要被刷新到磁盘上去（它们已在内存里更新，但是磁盘上的还没更新）。然而，第 10 行也显示出在那一刻并没有刷新操作在进行，这不成问题，因为 InnoDB 会在需要的时候刷新它们。

第 11 行显示了 InnoDB 已经读、创建和写了多少页。页读、写的值来自于从磁盘读入到缓冲池的数据，反之亦然。页创建的值来自于 InnoDB 在缓冲池里申请的页，不包括从数据文件读取出来的内容，因为它不关心内容是什么（例如，它们可能属于一个已被删除的表）。

第 13 行报告了缓冲池的命中率，它可以衡量 InnoDB 在缓冲池里找到所需要页的比例，是缓存效率的度量值。它的衡量范围是自最近一次 InnoDB 状态打印以来的命中率，因此，如果自那时以来，服务器一直很安静，那你就会看到“No buffer pool page gets since the last printout.”。因为 InnoDB 就是这么设计的，你不能直接拿 InnoDB 缓冲池的命中率跟 MyISAM 的关键缓冲区命中率作比较。

13.3.10 ROW OPERATIONS

这一段显示的是 ROW OPERATIONS 及其他形形色色的 InnoDB 统计信息：

```
1  -----
2  ROW OPERATIONS
3  -----
4  0 queries inside InnoDB, 0 queries in queue
5  1 read views open inside InnoDB
6  Main thread process no. 10099, id 88021936, state: waiting for server activity
7  Number of rows inserted 143, updated 3000041, deleted 0, read 24865563
8  0.00 inserts/s, 0.00 updates/s, 0.00 deletes/s, 0.00 reads/s
9  -----
10 END OF INNODB MONITOR OUTPUT
11 =====
```

第 4 行显示了 InnoDB 内核里有多少条线程（在 TRANSACTIONS 节里，我们已经谈到过这个）。位于队列里的查询就是 InnoDB 出于限制并发执行的线程数的目的，而还未将其置入内核的线程。这些查询在进入队列前也可以进入睡眠状态——我们在前文里已经讨论过这个了。

第 5 行显示了 InnoDB 已经打开了多少个读视图。一个读视图就是在事务开始之时取得的一张源自数据库内容的一致 MVCC “快照”。在 TRANSACTIONS 节里，你能查看到一个特定的事务是否拥有一个读视图。

第 6 行显示了内核里主线程的状态。在 MySQL 5.0.45 和 5.1.22 里，可能的状态值包括了下面这些：

- archiving log（如果日志归档功能已经开启的话）
- doing background drop tables
- doing insert buffer merge
- flushing buffer pool pages
- flushing log
- making checkpoint
- purging

578

- reserving kernel mutex
- sleeping
- suspending
- waiting for buffer pool flush to end
- waiting for server activity

第 7、8 行显示了行插入、更新、删除和读取次数的统计信息，以及这些数值在每一秒里的平均值。如果你想知道 InnoDB 正在完成多少工作，查看这些数值就是个不错的办法。

SHOW INNODB STATUS 的输出以第 9 到 13 行为结束。如果你没看到这几行文字，说明你的系统里可能有一个非常大的死锁存在，它截断了这些信息的输出。

13.4 SHOW PROCESSLIST

进程列表就是连接的列表，或者是当前正连接到 MySQL 的线程的列表。SHOW PROCESSLIST 列出这些线程时，也显示了线程的当前状态信息。举例来说：

```
mysql> SHOW FULL PROCESSLIST\G
***** 1. row *****
    Id: 61539
    User: sphinx
    Host: se02:58392
    db: art136
    Command: Query
    Time: 0
    State: Sending data
    Info: SELECT a.id id, a.site_id site_id, unix_timestamp(inserted) AS
inserted, forum_id, unix_timestamp(p
***** 2. row *****
    Id: 65094
    User: mailboxer
    Host: db01:59659
    db: link84
    Command: Killed
    Time: 12931
    State: end
    Info: update link84.link_in84 set url_to =
replace(replace(url_to, '&', '&'), '%20', '+'), url_prefix=repr
```

有几种工具（例如 innotop）可以为你显示进程列表的一个更新情况。

Command 和 State 列就是线程真正显示“状态”的地方。在本例中，第一条线程运行着一个查询，并且正在发送数据；第二条线程已被杀死了，可能是因为它已经执行了太长的时间却还没结束，有人就用 KILL 命令将其终止了。一条线程能处于这种状态好一会儿，因为 KILL 命令不是立即生效的。举例来说，它可能会花费一些时间回滚线程里的事务。

SHOW FULL PROCESSLIST（这里加上了 FULL 关键字）会显示每一条查询的全部文本，在这里只是截取了前面的 100 个字符。

13.5 SHOW MUTEX STATUS

SHOW MUTEX STATUS 返回的是 InnoDB 互斥量的细节信息，通常是用于深入观察系统的伸缩性和并发性的问题。在代码里面，每一个互斥量保护着一块临界区，这在上文里已经说明过了。

这些输出信息会随着不同的 MySQL 版本和编译选项而有所不同。有时，你能得到互斥量的名称和每一个互斥量的一些列的信息；有时，你只能得到一个文件名、一条线和一个号码。你需要编写一个脚本来聚合这些输出信息，因为输出信息会比较庞大。以下是仅有一个互斥量的输出示例：

```
***** 1. row *****
Mutex: &(buf_pool->mutex)
Module: buf0buf.c
Count: 95
Spin_waits: 0
Spin_rounds: 0
OS_waits: 0
OS_yields: 0
OS_waits_time: 0
```

你可以使用这些输出信息来确定 InnoDB 的哪一部分已经成为了瓶颈，例如多 CPU 也会引起瓶颈。最近，MySQL 已经修复了很多在多 CPU 系统里的伸缩性问题，但是，跟互斥量相关的一些问题还没解决。比较典型的一个就是一些人遇到过的 AUTO_INCREMENT 锁，它是每张表一个，在表内它属于全局的，在 InnoDB 和插入缓冲区里有一个互斥量保护着它。哪里有关互斥量，哪里就有潜在的竞争。

输出里的列信息说明如下：

Mutex

互斥量的名称。

Module

互斥量在源代码文件中定义的位置。

Count

互斥量被请求过的次数。

Spin_waits

InnoDB 采用循环等待的方式来获取互斥量的次数。你会回忆起 InnoDB 首先会采用循环等待的方式，然后再回到一个操作系统等待。

Spin_rounds

在 InnoDB 循环等待时，它检查互斥量是否已被释放的次数。

OS_waits

InnoDB 返回到操作系统等待以获取互斥量的次数。

OS_yields

线程把互斥量让给其他线程的次数。

OS_waits_time

如果 timed_mutexes 设为 1，这就是已经花费的等待时间，以毫秒为单位。

比较这些计数器的相对大小，你就能发现热点所在。这里有 3 种办法来缓解瓶颈问题：避开 InnoDB 的虚弱点、限制并发、或者平衡 CPU 密集的循环等待和资源密集的操作系统等待。关于如何调优 InnoDB 并发的更多内容，

13.6 复制的状态

MySQL 有几个命令是用于监控复制的。在一个主服务器上，SHOW MASTER STATUS 可以显示出主机的复制状态和配置：

```
mysql> SHOW MASTER STATUS\G
***** 1. row *****
      File: mysql-bin.000079
      Position: 13847
      Binlog_Do_DB:
      Binlog_Ignore_DB:
```

181 输出信息包括了主服务器上当前二进制日志的位置。使用 SHOW BINARY LOGS 可以得到一个二进制日志的清单：

```
mysql> SHOW BINARY LOGS
+-----+
| Log_name          | File_size |
+-----+
| mysql-bin.000044 |    13677 |
...
| mysql-bin.000079 |    13847 |
+-----+
36 rows in set (0.18 sec)
```

要想看二进制日志里的事件，就要使用 SHOW BINLOG EVENTS 了。

在一个从服务器上，你可以使用 SHOW SLAVE STATUS 来查看从服务器的状态和配置信息。这里就不把它的输出内容示例包括进来了，因为它们有一点冗长，但是，我们会说明其中的一些要点。首先，你能看到从服务器的 I/O 和 SQL 线程的状态，包括其中的任何错误；其次，你能看到这个从服务器在复制上落后主机多少时间；最后，出于备份和克隆从服务器的目的，有另外三套二进制日志的定位坐标放入输出内容里：

Master_Log_File/Read_Master_Log_Pos

I/O 线程在主服务器二进制日志上正在读取的位置。

Relay_Log_File/Relay_Log_Pos

SQL 线程在从服务器转发日志上正在执行的位置。

Relay_Master_Log_File/Exec_Master_Log_Pos

SQL 线程在主服务器二进制日志上正在执行的位置。它跟 Relay_Log_File/Relay_Log_Pos 在逻辑位置上是一样的，但是，前者在从服务器的转发日志里，而后者是在主服务器的二进制日志里。换句话说，如果你分别到两个日志里查看对应的位置，会发现那一处的日志事件是相同的。

13.7 INFORMATION_SCHEMA

INFORMATION_SCHEMA 数据库里是一套按照 SQL 标准定义的系统视图。MySQL 实现了许多个标准视图，还增加了其他一些视图。在 MySQL 5.1 里，许多视图对应于 MySQL 的 SHOW 命令，例如，SHOW FULL PROCESS LIST 和 SHOW STATUS，但是，也有一些视图没有可对应的 SHOW 命令。

INFORMATION_SCHEMA 视图的美感在于你可以使用标准的 SQL 查询它们,由此带来的兼容性超过了 SHOW 命令,SHOW 命令只能生成结果,无法再聚合、联接,更无法用标准 SQL 来操作。有了系统视图里的所有这些可用数据,你就能编写出一些有趣又有用的查询。

举例来说,你想知道 Sakila 示例数据库里有哪些表引用了 actor 表吗?一致的命名约定使这个问题显得相对比较简单:

```
mysql> SELECT TABLE_NAME FROM INFORMATION_SCHEMA.COLUMNS
-> WHERE TABLE_SCHEMA='sakila' AND COLUMN_NAME='actor_id'
-> AND TABLE_NAME <> 'actor';
+-----+
| TABLE_NAME |
+-----+
| actor_info |
| film_actor |
+-----+
```

我们要找出本书几个示例中哪些有多列索引的表,以下就是针对它的查询:

```
mysql> SELECT TABLE_NAME, GROUP_CONCAT(COLUMN_NAME)
-> FROM INFORMATION_SCHEMA.KEY_COLUMN_USAGE
-> WHERE TABLE_SCHEMA='sakila'
-> GROUP BY TABLE_NAME, CONSTRAINT_NAME
-> HAVING COUNT(*) > 1;
+-----+-----+
| TABLE_NAME | GROUP_CONCAT(COLUMN_NAME) |
+-----+-----+
| film_actor   | actor_id,film_id         |
| film_category | film_id,category_id      |
| rental       | customer_id,rental_date,inventory_id |
+-----+-----+
```

你也能写出更加复杂的查询,就像你基于任何普通表上做的那样。MySQL Forge (<http://forge.mysql.com>) 就是查找和分享这些查询语句的好地方,它拥有很多实例,例如找出重复的或者冗余的索引、用低基数找到索引、以及其他很多很多的查询语句。

这些视图的最大缺点就是跟 SHOW 命令相比,查询它们有时会非常慢。它们典型的工作方式是获取所有数据,存放在一个临时表内,然后在临时表上执行查询。对于许多监控、故障排除和调优等用途而言,与其键入完整的 SQL 语句从视图里取得数据,还不如直接键入 SHOW 命令来得快。

在本书编写的时候,这些视图还是无法被更新的。虽然你可以从它们那里获取到服务器的设置信息,但是,你无法更新它们,也无法从这个途径更改服务器的配置。在实际使用过程中,这些限制条件意味着你还是要使用 SHOW 和 SET 命令来配置服务器,哪怕是 INFORMATION_SCHEMA 的视图对于其他任务非常有用。

用于高性能 MySQL 的工具

Tools for High Performance

MySQL 服务器的发布包没有包含那些能完成许多常见任务的工具，例如监控服务器的工具、比较服务器间数据的工具。所幸的是，MySQL 志愿者社区已经开发了多种多样的工具，帮你解决各种问题。许多公司也提供了商业化的替代工具或是对 MySQL 已有工具的补充。

本章内容将遍及一些最常用最重要的产品化 MySQL 工具。我们把这些工具分成以下几类：界面、监控、分析和辅助工具。

14.1 带界面的工具

Interface Tools

带界面的工具可以帮你运行查询、创建表和用户、完成其他常见的任务。这一节里将简要地描述一下此类工具里最常用的几个。这些工具能做的事情，你用 SQL 查询和命令也是能做到的，但是使用它们会更加方便，即能避免错误，又能加快你的工作进度。

14.1.1 MySQL 可视化工具

MySQL Visual Tools

MySQL AB 发布了一套可视化工具，其中包括了 MySQL 查询浏览器（MySQL Query Browser）、MySQL 管理员（MySQL Administrator）、MySQL 迁移工具箱（MySQL Migration Toolkit）和 MySQL 工作台（MySQL Workbench）。这些工具都可以免费使用，你可以把它们一起下载下来安装在电脑上。它们能在所有流行的操作系统上运行，此前其中还带有一些恼人的小问题，但是，MySQL AB 已经找出并修复了这些 bug。

MySQL 查询浏览器可以用于运行查询、创建表和存储过程、导出数据和浏览数据库结构，它的使用帮助已经集成在 MySQL 的 SQL 命令和函数手册里。对于那些开发和查询 MySQL 数据库的人来说，MySQL 查询浏览器是最有用的工具。

584 MySQL 管理员的功能集中在服务器管理上，所以，它最适合 DBA 使用，而不是开发人员和分析人员。它可以帮助 DBA 把创建备份、创建用户并分配权限、显示服务器日志和状态信息等过程进行自动化处理。它还包括了一些基本的监控功能，例如图形化的状态变量显示，但是它没有下文里会提到的交互式监控工具那么灵活，也无法为后续的分析工作记录当前的系统统计信息——这对其他许多监控工具来说是都能做到的。

这个工具集里还包括了 MySQL 迁移工具箱，它可以帮你把数据从别的数据库系统迁移到 MySQL 里。工具集里的 MySQL 工作台是一个建模工具。

MySQL 自带工具的好处是它们是免费的，而且质量也相当不错，能够在许多桌面操作系统上运行。它们有一套简洁的功能特性，可以胜任许多任务。其中最突出的功能就是 MySQL 管理员里的用户管理和备份，以及 MySQL 查询浏览器里的集成文档。

这些工具的主要缺点是稍微有点简单，如果少一些华而不实的功能，可能会更合乎高级用户的需求。关于这个工具集的完整的描述，包括界面截图，你可以在 MySQL 的网站 <http://www.mysql.com/products/tools/> 上找到。



提示：MySQL 工作台最近已经被从头改写了一遍，并推出了免费的和商用的两个版本。免费版的功能并没有缩水，但是商业版里包含了一些插件，可用于一些任务的自动化，减少了手工操作。在写下这段话的时候，MySQL 工作台的版本仍然是 beta 版。

14.1.2 SQLyog

SQLyog 是最常用的 MySQL 可视化工具，它设计良好，专门支持 DBA 和开发人员的工作。罗列它全部功能特性的清单会很长，所以就不包括进来了，下面只提一下其中的“亮点”：

- 代码自动完成能帮你更快地写查询语句。
- 能够通过 SSH 隧道连接到远程主机服务器。
- 可视化工具和向导可以帮你像构建查询语句一样完成普通的任务。
- 可以进行任务调度，在预定时间里执行备份、数据导入和数据同步等操作。
- 有键盘快捷键功能。
- 架构比较，它提供了访问对象（例如表、视图）属性的功能。
- 用户管理。

SQLyog 拥有你所期望的所有标准功能，例如配置编辑器。这个工具只能在 Microsoft 的 Windows 平台上使用，⁵⁵ 全功能版的需要出钱购买，有限功能版是免费的。更多关于 SQLyog 的信息可以访问 <http://www.Webyog.com>。

14.1.3 phpMyAdmin

phpMyAdmin 是一款很流行的管理工具，它在一个 Web 服务器上运行，让你可以通过基于浏览器的界面来管理你的 MySQL 服务器。在查询和管理方面，它有着许多非常出彩的功能特性。它主要的优点是平台独立性、一个庞大的功能集，以及可以通过浏览器访问。当你远离服务器环境并且只有一个浏览器可供你使用时，基于浏览器访问的功能会是非常有用的。举例来说，你可以在一台只拥有 FTP 访问权的主机服务器上安装上 phpMyAdmin，而原本连 mysql 客户端或其他 shell 程序都无法运行。

phpMyAdmin 是个很轻便的工具，在各种情况下都能找出满足你所需要的功能。当你在一个可以提供 Web 访问的系统上安装 phpMyAdmin 的时候一定要小心，因为如果你的服务器安全无法保证，那就是给攻击者提供了一条便捷的入侵路径。

但是，也有 phpMyAdmin 的反对者们声称 phpMyAdmin 的功能太多太庞大了，错综复杂。phpMyAdmin 常驻在 SourceForge.net 上，在那里它一直是最优秀的项目之一。更多信息可以访问 <http://sourceforge.net/projects/phpmyadmin/>。

14.2 监控工具

MySQL 的监控可以作为一个独立的主题写成一本书：它是一个很大很复杂的任务，不同的应用有着不同的需求。但是，我们可以针对这个标题把一些很好的工具和资源介绍给你。

“监控”是被大家滥用的术语之一，承载了几重意思。但是，在我们的经验里，多数基于 MySQL 的购物网站需要做许多不同类型的监控任务。

我们讲到的监控工具被分为非互动的和互动的两类。非互动监控常常就是一个自动化系统，它接收系统的测量值，如果有超出安全范围的，就通过发出警告提醒管理员。互动监控工具可以让你实时地观测服务器。在下面几个小节里，我们会把这两类工具分开来讲。

你可能对监控工具在其他方面的差别也很感兴趣，例如被动监控和主动监控，后者会发送警报信息并作出初步反应（就像 Nagios 一样）；或者你可能正在寻找这样一个工具：它可以创建一个信息仓库，而不仅仅是显示当前的统计信息。在接下来的介绍里，我们将会逐个指出每一种工具的特点。

586 14.2.1 非交互式监控系统

有许多监控系统不是专为监控 MySQL 而设计的，它们就是一个通用系统，里面设计了一个周期性的任务，定时去检查各类资源的状态，例如像服务器、路由器、以及各种软件（包括 MySQL）。它们常常会提供一个插件架构，同时有一个预定的 MySQL 插件可供使用。这样的一些系统能够记录监控对象的状态，并通过 Web 界面用图形化的形式表示出来。当监控对象出现问题，或者状态值超过安全范围时，它们还能发送报警信息，或者执行一个初始化的动作。

一般而言，你要把这个监控系统安装在自己的服务器上，通过它去监控其他服务器。如果你正在用它监控着一台重要的服务器，那么它很快就会成为你的 IT 基础架构里关键的一部分，因此，你还要采取一些额外的步骤，例如监控系统本身也必须是冗余的，以便于故障恢复。

一个自动化的监控系统能记录历史信息，当一个 MySQL 实例因为负载增加或者遇到其他故障时，监控系统显示出的状态变化趋势会成为“救命恩人”。修复问题常常需要知道哪些信息发生了变化，这就要求了解你的服务器的历史信息。当监控系统发现有些信息不对头时，它会在灾难发生前及时警告你，帮你集中精力发现并修复这个即将发生的故障。

自己开发的系统

许多组织在开始时都是自己构建监控报警系统。当只有几个系统需要监控，也没几个人使用时，它们往往都能工作得很好。然而，当组织逐渐扩大，变得更复杂，越来越多的系统管理员加入进来后，自产的监控系统就会趋于崩溃。每次当网络发生故障时，它就会发送出数以千记的消息把邮箱塞满；或者当有关键性问题发生时，它就悄悄地崩溃了，也不提醒任何人。对于自产的监控系统，重复或冗余的通知信息是最常见的问题，会妨碍到正常工作的完成。

如果你正打算自己编写一个监控工具——哪怕只要求像 cron 那么简单的功能：检查一个查询，如有问题就用邮

件通知某人，那么，你应该仔细考虑下面这些建议。最好还是把时间和精力花在学习使用下文中要提到的监控系统之一上。即使这些系统里的某几个有着陡峭的学习曲线，看起来不值得初始投资，但是，从长远来看，它们可以帮你节省时间和精力，使你组织里的系统状态渐入佳境。使用它们中的一个，即使开始时会很糟糕，但最终会被证明这是比实现自产系统更合适的选择。至少从长远来看，你在使用一个标准的监控系统时将获得经验和技能。

Nagios

Nagios (<http://www.nagios.org>) 是一个开源的监控报警系统，它会定时检测你定义的服务，并将检测结果跟默认值或者上下限作比较。如果结果值超出了范围，Nagios 就会执行一段程序并且/或者把这个问题提醒某人。Nagios 的联系人和报警系统可以让你把通知发给不同的联系人。根据日期时间或其他条件，你也可以更改警报，或者把警报发送到其他地方，或者预告停机时间。Nagios 也理解服务之间的依赖关系，因此，当 MySQL 实例无法启动时，如果原因是网络中间的路由器故障或主机服务器崩溃而使服务器不可到达的话，Nagios 就不会来打扰你。

Nagios 可以将任何可执行文件作为插件来运行，只要给予它正确的参数，就能输出正确的结果。因此，Nagios 的插件可以是各种语言编写的，包括 shell 语言、Perl、Ruby 和其他脚本语言。<http://www.nagiosexchange.org> 站点就是 Nagios 专门的共享插件和分类站点。如果你在上面找不到你所需要的插件，自己创建一个也很简单：只要能接受标准的参数，并有正确的退出状态就可以，你也可以有选择地加入一些能让 Nagios 捕获的输出信息。

在许多操作系统上，Nagios 通过几种方法（包括主动检查、远程执行插件和被动检查——主要是接收其他系统“推”过来的状态数据），几乎可以监控你能测量到的一切状态。它还带有一个 Web 界面，通过这个界面，你可以检查状态、显示图表、可视化你的网络和当前的状态、安排计划中的停机时间等等。

Nagios 主要的缺点在于它让人望而生畏的复杂性。哪怕你一时学得很好，但到用的时候还是有点难度。它还把所有的配置项目都放在文件里，每个配置项目的语法都很特别，很容易出错。当你的系统日渐庞大的时候，更改这些配置信息会是一件艰苦的体力活。最后，它的图表、趋势图和可视化功能都相当有限。Nagios 把一些性能信息和其他数据存储在一个 MySQL 服务器上，然后可以基于这些数据来生成图表，但是，对于其他系统，兼容性还是不够的。

有几本专门讲述 Nagios 的书，我们比较喜欢 Wolfgang Barth 的《Nagios System and Network Monitoring》(No Starch Press)。

可替代 Nagios 的工具

虽然 Nagios 是最常用的多功能监控报警软件（注 1），但是，还是有其他几款开源工具可供你选用：

Zenoss

Zenoss 是用 Python 编写的，拥有基于浏览器的用户界面，并使用了 Ajax 使操作更加快捷而富有效率。它将监控、报警、趋势显示、图表显示和记录历史数据等功能合成在一个统一的工具里，它还能在网上自动发现资源，在默认情况下，Zenoss 使用 SNMP 从远程机器上收集数据，但它也可以使用 SSH，并且支持 Nagios 插件。想要了解更多相关信息，可以访问 <http://www.zenoss.com>。

注 1：可能就是因为你一旦你安装、配置好 Nagios，你就永远不会再想到监控系统了。

Hyperic HQ

Hyperic HQ 是一款基于 Java 的监控系统，它的目标跟其他同类别的软件不太一样，它要成为企业级的监控系统。跟 Zenoss 一样，它也能自动发现资源，支持 Nagios 插件，但是它的逻辑组织和架构很不一样，显得有一点庞大。至于它是不是适合你的需要，那要看你的参数设置和监控的方式了。更多关于 Hyperic HQ 的信息，可以访问 <http://www.hyperic.com>。

OpenNMS

OpenNMS 是由 Java 编写的，拥有一个活跃的开发社区。它具备了常规的功能，例如监控和报警，也加入了图表和趋势显示的功能。它的目标是高性能、伸缩性、自动化以及良好的兼容性。跟 Hyperic 一样，它也企图成为一款企业级的监控软件，可以用在大型的关键系统上。更多关于 OpenNMS 的信息，可以访问 <http://www.opennms.org>。

Groundwork Open Source

Groundwork Open Source 实际上是基于 Nagios 的，它把 Nagios 和其他几个工具集成为一个系统，并安上一个统一的门户界面。描述它的最好方法可能就是：如果你对 Nagios、Cacti 及其他工具很熟悉，并且能够花大量的时间把它们无缝地集成在一起的话，你也能在家庭作坊里做一个出来。关于它的更多信息，可以访问 <http://www.groundwork-kopendsource.com>。

Zabbix

Zabbix 是一个开源监控系统，在许多方面跟 Nagios 很相像，但是也有一些关键的不同点。例如，它把所有配置信息和其他数据都存放在一个数据库里，而不是放在配置文件里；它比 Nagios 存储了更多类型的数据，这样可以生成更好的趋势图和历史报告。它的网络图表和可视化功能也优于 Nagios。很多使用它的人发现它更易于配置，更具有兼容性。说起来它也能比 Nagios 承受更大的负载。在另一方面，Zabbix 的社区要比 Nagios 少，它的报警功能也不够高级。更多关于 Zabbix 的信息，可以访问 <http://www.zabbix.com>。

MySQL 监控和建议服务

MySQL 自己的监控方案就是设计用来监控 MySQL 实例的，但也能够监控主机的一些关键方面。这个工具不是开源的，需要 MySQL 企业订阅费。

589 与 Nagios 相比，这个服务的主要优点在于它提供了一套预设的规则或者说是“顾问”，通过这些规则，它会去检查服务器性能、状态和配置等许多方面。当它注意到有问题发生时，它会向用户建议解决方案，而不仅仅是向管理员报告说发生了什么错误。它也有一个设计良好的仪表板式界面，能够即刻显示所有服务器的状态信息。

尽管采用 Nagios 或其他系统来监控可能也可以获得到同样的统计数据，但是，对于 Nagios 来说，要想获取到 MySQL 里大量的测量值，它还要增加许多工作量用于编写插件、修改配置之后才能做到。而对于 MySQL 监控和建议服务，这些都是唾手可得的。

这个产品也有一个缺点，就是你无法使用它去监控网络的其余部分，它只是用来监控 MySQL 的。而且，它还要在每一台被监控的服务器上安装一个代理，一些 MySQL 管理员对此会很反感，他们希望服务器除了核心服务外，越简单越好。

更多信息，可以访问 <http://www.mysql.com/products/enterprise/advisors.html>。

MONyog

MONyog (<http://www.Webyog.com>) 是一个轻量级的无代理的监控系统，它跟以上那些工具有着不同的实现方

法：它的底层是一个 JavaScript 引擎，所有配置都是通过 JavaScript 对象模型来完成的。它被设计为在桌面系统上运行，运行时它会在一个闲置的端口上打开一个 HTTP 监听器。这样，你就可以把你的浏览器指向这个端口，查看到 MySQL 服务器的信息了，这些信息都是结合了 Javascript 和 Flash 来表示的。

MONyog 实际上有交互式和非交互式两种类型，因此，你可以把这两种类型的监控功能都尝试着用用看。

基于 RRDTool 的系统

严格地说，RRDTool (<http://www.rrdtool.org>) 不算是一个监控系统，但是，它很重要，有必要在此提一下。很多组织里都是使用几种脚本或程序——这些一般都是自制的——从服务器那里读取到信息，然后再保存到循环数据库（Round-robin database，RRD）文件里。在许多要获取记录生成图表的环境下，RRD 文件是一个很适合的解决方案。它们能自动聚合输入的数据，如果输入数据值没有按期在随后提交进来时，还能在随后插入这些丢失的数据。它们还都带有强大的图表工具，能够生成漂亮的与众不同的图表。现在已经有一些基于 RRDTool 的系统可供使用了。

Multi Router Traffic Grapher，或者叫 MRTG (<http://oss.oetiker.ch/mrtg/>) 就是一款典型的基于 RRDTool 的系统。它真正的设计初衷是记录网络数据流，但是它也被扩展用来记录和图表化表示其他一些东西。

Munin (<http://munin.projects.linpro.no>) 是一个能为你采集数据的系统，将它放入 RRDTool 后，就会根据数据生成不同粒度的图表。它能从配置信息里生成静态的 HTML 文件，这样你就可以轻松地浏览，查看趋势情况。要定义一个图表也很容易，只要你创建一个插入式脚本，它的命令行帮助输出结果需要有 Munin 能识别的特殊语法，Munin 把这些参数用作图表指令。Munin 的缺点包括它要求每个被监控的系统都要安装一个代理、简化了的“一个尺寸、大小通用”的配置方式和图表选项对于某些需要来说还不够有弹性。

Cacti (<http://www.cacti.net>) 是另外一个常用的图表和趋势显示系统。它的工作方式是：从系统里获取数据，然后保存在 RRD 文件里，然后用 PHP Web 界面的形式，使用 RRDTool 把数据以图表的形式展示出来。这个显示界面也是配置和管理界面（配置信息存储在一个 MySQL 服务器里）。它是模板驱动的，因此，你可以自己定义模板，并放到你的系统里使用。它也可以从 SNMP 或者其他自定义脚本里获取到数据。

Cricket (<http://cricket.sourceforge.net>) 是一个用 Perl 编写的跟 Cacti 类似的系统，使用的是基于文件的配置系统。Ganglia (<http://ganglia.sourceforge.net>) 也跟 Cacti 类似，但它的设计初衷是用于监控集群和系统网络，因此，你可以查看到由许多服务器信息聚合得到的结果，也可以按照你的意愿，查看单独某台服务器的信息。（Cacti 和 Cricket 无法显示聚合数据。）

以上这些系统都可以被用作 MySQL 系统信息的收集、记录、图表化数据和报告，它们在用途方面差异较小，都具备了不同程度的兼容性。但是，它们缺乏真正意义上的兼容性，比如当某些东西出错时，它要能够有针对性地发送警报信息给某些人。它们中的一些甚至没有“错误”的概念。所以，有些人把这一点看作是此类系统的一大缺点，觉得最好还是把记录、图表化表示、报警这几项功能都独立开来。事实上，Munin 特地设计了使用 Nagios 来作为它的报警系统。然而，对于其他几个来说，这的确是缺点。另外还有一个缺点就是安装和配置这样一个系统，使其能完全满足你的需求，须投入很多时间和努力，不过，这一点也并不会总这样。

最后，你应该考虑到将来的需求。RRD 文件无法让你使用 SQL 或其他标准方法来查询它里面的数据。而且，在默认情况下，它永远会以一种恰好的粒度来存储数据，许多 MySQL 管理员就不愿意接受这种限制，转而选择一个关系数据库来存储这些历史数据。许多 DBA 也希望能以更个性化更有弹性的方式来记录数据，所以，他们转而编写他们自己的系统或修改一个现有的系统。

基于 RRDTool 的系统是不是正好能与你的组织相匹配，这还是要看个人的选择、管理系统所需的可用技能，以及你组织的具体需求。

591 14.2.2 交互式工具

交互式工具就是那些在你需要时就可以启动起来，并以视图显示的形式不断获取最新的服务器状态的软件。接下来，我们将主要讲解 innotop (<http://innotop.sourceforge.net>)，但是，也有别的几个工具可供使用，例如 mtop (<http://mtop.innotop.sourceforge.net>)、mytop (<http://jeremy.zawodny.com/mysql/mytop/>) 及一些基于 Web 的 mytop 克隆版本。

innotop

本书的作者之一——Baron Schwartz 编写了 innotop。别去理会它的名称，实际上它不仅仅用于监控 InnoDB 的内部信息。这个工具的设计灵感来自于 mytop，但是它提供了更多的功能。它有许多模式可用来监控所有类型的 MySQL 内部状态，包括 SHOW INNODB STATUS 里的全部可用信息，它能将这些信息分解成不同的部分。它也能让你同时监控多个 MySQL 实例，极具可配置性和可扩展性。

它的功能特性包括以下这些：

- 事务列表显示 InnoDB 当前的全部事务；
- 查询列表显示当前正在运行的查询；
- 显示当前锁和锁等待的列表；
- 服务器状态和变量的摘要信息显示了数值的相对变化幅度；
- 有多种模式可用来显示 InnoDB 内部信息，例如缓冲区、死锁、外键错误、I/O 活动情况、行操作、信号量，以及其他更多的内容；
- 复制监控，将主机和从服务器的状态显示在一起；
- 有一个显示任意服务器变量的模式；
- 服务器组可以帮你更方便地组织多台服务器；
- 在命令行脚本下可以使用非交互式模式。

innotop 很易于安装，你可以将它从操作系统的 package 库里取出来安装，也可以从 <http://innotop.sourceforge.net> 上下载到本地，然后解压缩，运行标准的 make 安装过程：

```
perl Makefile.PL
make install
```

一旦你安装完成，就可以在命令行里执行 innotop，它会带你完成连接到 MySQL 实例的过程。它会自动去读取你的 ~/.my.cnf 选项文件，这样，你除了输入服务器的主机名和按几次 Enter 键之外，什么都不用做。连接完成以后，就处在 T 模式（InnoDB Transaction）里了，这时，你应该看到 InnoDB 事务列表，如图 14-1 所示。

```

Terminal - baron@keywest:~
InnoDB Txns (? for help) srvr 1, 25+21:37:41, InnoDB 2s :-), 42.87 QPS, 21 thd,
CXN  History  Versions  Undo  Dirty Buf  Used Bufrs  Txns  MaxTxnTime  LStrcts
srvr_1      44      169  0 0      25.73%    94.36%    13      49:26      0

CXN  ID      User  Host  Txn Status  Time  Undo  Query Text
-----

```

图 14-1: innotop 里的 T (Transaction) 模式

默认情况下, innotop 采用过滤器来减少零乱的信息 (就是 innotop 能显示出的所有信息, 你可以定义自己的过滤器, 或者定制它内部的过滤器)。在图 14-1 里, 大多数事务都已经被过滤掉了, 只显示出当前活动的事务。你可以按 i 键关闭过滤, 让数量众多的事务信息把整个屏幕填满。

Innotop 在这个模式下会显示一个头部信息和一个主线程列表。头部信息里是一些 InnoDB 的总体信息, 例如历史清单的长度、还未清除的 InnoDB 事务的数目、脏缓冲区在缓冲区池里所占的百分比等等。

这时你要按的第一个键应该是问号 (?), 查看一下帮助信息。虽然在屏幕上显示出的帮助内容会根据当前不同的模式而不同, 但是, 每一个活动的键总是会显示出来, 这样, 你就能看到所有可执行的动作了。图 14-2 显示的是 T 模式下的帮助信息。

我们在这里不会讲到所有的模式, 但是, 你从帮助信息里就能看出, innotop 有许多多功能特性。

唯一要提及的是一些基本的自定义功能, 告诉你如何监控你想要监控的信息。innotop 的强大功能之一就是能够解读用户定义的表达式, 例如 Uptime/Questions 的意思是每秒钟的查询数, innotop 会显示自服务器启动以来和/或自上次采样之后累加起来的结果值。

于是, 往显示表格里添加你自己的列就方便很多。举例来说, 在 Q (Query List) 模式下, 有一个头部信息能显示出服务器的一些总体信息。让我们看看怎么将它修改一下, 使它能显示出索引键缓存有多满。启动 innotop, 按下 Q 键进入 Q 模式。这时的操作结果看起来像图 14-3 一样。

这个屏幕截图已有过删减, 因为在这个练习里, 我们对查询列表没有兴趣。我们只关心头部信息。

```

Terminal - baron@keywest:~
InnoDB Txns (? for help) srvr 1, 25+21:44:21, InnoDB 10s :-), 32.47 QPS, 21 thd,

Switch to a different mode:
B InnoDB Buffers      M Replication Status  S Variables & Status
D InnoDB Deadlocks    O Open Tables         T InnoDB Txns
F InnoDB FK Err       Q Query List          W InnoDB Lock Waits
I InnoDB I/O Info     R InnoDB Row Ops

Actions:
a Toggle the innotop process      k Kill a transaction's connection
c Choose visible columns          n Switch to the next connection
d Change refresh interval         p Pause innotop
e Explain a thread's query        q Quit innotop
f Show a thread's full query      r Reverse sort order
h Toggle the header on and off    s Change the display's sort column
i Toggle inactive transactions    x Kill a query

Other:
TAB Switch to the next server group  / Quickly filter what you see
! Show license and warranty          @ Select/create server connections
# Select/create server groups        \ Clear quick-filters
+ Edit configuration settings        ^ Edit the displayed table(s)

Press any key to continue

```

图 14-2: innotop 帮助信息

| CXN | When | Load | QPS | Slow | QCacheHit | KCacheHit | BpsIn | BpsOut |
|--------|-------|------|--------|--------|-----------|-----------|---------|---------|
| srvr_1 | Now | 0.01 | 40.47 | 0 | 53.52% | 100.00% | 135.48k | 319.85k |
| srvr_1 | Total | 0.00 | 140.26 | 11.91k | 6.02% | 96.33% | 110.58k | 872.50k |

| CXN | ID | User | Host | DB | Time | Query |
|-----|----|------|------|----|------|-------|
| | | | | | | |

图 14-3: Q(Query List)模式时的 innotop

头部显示的是“现在”（它衡量的是自上次 innotop 从服务器取得新数据之后的增长情况）和“总共”（它衡量的是自 MySQL 启动后 25 天来的总体活动情况）的统计信息。头部里的每一列都是源自 SHOW STATUS 和 SHOW VARIABLES 里对应变量的值。像图 14-3 显示的头部是内建的，但是，要添加你自己的列也很容易。所有你要做的就是头部“表格”里增加一列：按下[^]键打开表格编辑器，然后，在参数输入框里输入 q_header 开始编辑头部表格（图 14-4）。Tab 补全操作是内建的，因此，你可以按 q 然后再按 Tab 来完成整个词的输入。

在此之后，你将看到 Q 模式下头部表格的定义（图 14-5）。这个表格定义显示了表格的各个列。第 1 列是选择记号，我们可以移动这个选择记号，然后进行记录、编辑及其他操作。

```

Choose from
processlist MySQL Process List
q_header Q-mode Header

Choose a table: q_header

```

图 14-4: 增加一个头部信息（开始）

（按下[?]可以查看到整个列表），但是，此时我们只要创建一个新的列：只须按下 n 键，然后输入新列的名称（图 14-6）即可。

| name | hdr | label | src |
|----------------|-----------|--------------------------------|--------------------|
| > cxn | CXN | Connection from which the data | cxn |
| when | When | Time scale | when |
| load | Load | Server load | \$cur->{Threads_co |
| qps | QPS | How many queries/sec | Questions/Uptime_h |
| slow | Slow | How many slow queries | Slow_queries |
| q_cache_hit | QCacheHit | Query cache hit ratio | (Qcache_hits 0)/(|
| key_buffer_hit | KCacheHit | Key cache hit ratio | 1-(Key_reads/(Key_ |
| bps_in | BpsIn | Bytes per second received by t | Bytes_received/Upt |
| bps_out | BpsOut | Bytes per second sent by the s | Bytes_sent/Uptime_ |

图 14-5: 增加一个头部信息（选择）

```

Choose a name for the column. This name is not displayed, and is only used for
internal reference. It can only contain lowercase letters, numbers, and
underscores.

Enter column name: kc_used

```

图 14-6: 增加一个头部信息（对列进行命名）

下一步就是输入列的头部名称，它会出现列的顶端（图 14-7）。最后，选择列的来源，这是一个表达式，innotop 会将它编译成一个内部函数，你可以使用来自 SHOW VARIABLES 和 SHOW STATUS 的变量名。我们使用了一些括号和 Perl 式的“或”默认用来避免被零除，因此，等式看起来会比较直观。这里，我们也使用了一个名叫 percent() 的 innotop 转换函数，把结果值格式化为百分比的形式，更多的用法可以查看 innotop 文档。图 14-8 显示了的就是这样一个表达式。

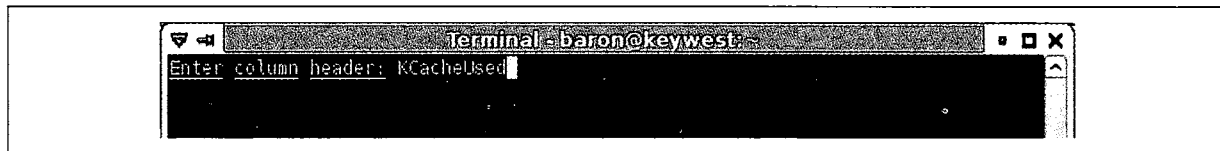


图 14-7：增加一列（列的文本标题）



图 14-8：增加一列（要计算的表达式）

按 Enter，你就会看到表格的定义跟原先一样，但是在底部多了新增的一列。按住+键一会儿，把它从列表的底部移上去，到 key_buffer_hit 列的旁边，然后再按 q 退出表格编辑器。瞧：你新增的列就在 KcacheHit 和 BpsIn 之间（图 14-9）。要定制 innotop 监控你想要的东西也是非常容易的，即使有它做不到的事情，你也可以通过编写插件来实现。在 <http://innotop.sourceforge.net> 上有更多相关文档可供参考。

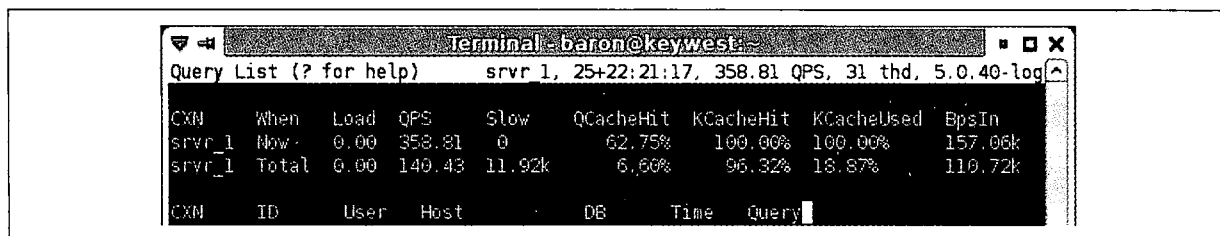


图 14-9：增加一列（最终效果）

14.3 分析工具

Analysis Tools

分析工具可以帮你自动化那些单调乏味的工作，例如监测服务器，找出还可以优化和调优的功能区域。这些工具可以作为解决性能问题的良好开端。如果它们中的一个发现了一个明显的问题，你就可以在这问题上全力以赴，也许能因此更快地解决它。

14.3.1 HackMySQL 工具

HackMySQL Tools

Daniel Nichter 维护着一个名叫 HackMySQL 的网站，在那里他提供了几款有用的 MySQL 工具。Mysqreport 是一个 Perl 脚本，它能检查 SHOW STATUS 的输出结果，把它们转换为易读的报告，并打印出来。阅读这个报告要比你自己检查 SHOW STATUS 要快很多，而且信息更全面。

下面是这份报告里各主要部分的概述，以 3.23 版本为例：

- “Key” 段显示的是你的键（索引）的使用情况。如果这些值不太“健康”，你就可能需要调优一下你的索引键缓存的设置了。
- “Questions” 段显示的是服务器正在执行的是哪一类查询，它可以帮你找出系统的负载主要集中在何处。
- “SELECT and Sort” 段显示的是服务器最常执行的是哪几种查询计划和排序策略。通过这一节信息，你可以找出索引问题或者那些不够优化的查询。
- “Query Cache” 段显示了你的查询缓存的使用情况。如果它有问题存在，那你应该调优它的设置，否则，你的工作负载就无法利用缓存的好处了，甚至会停掉缓存。
- 还有一些段显示了表、锁、连接和网络数据流的信息。这里如果有问题，一般就是表明服务器还有需要调优的地方。
- 还有 3 个段显示的是 InnoDB 性能度量值和设置参数。此处的问题可能是源自不当的服务器设置、硬件故障、查询或式样缺少优化等原因。

在 <http://hackmysql.com/mysqlreport> 上有更多此类内容，包括一份怎么解读报告的详尽指导。特别是你如果要经常在不熟悉的服务器上排除故障，花一些时间学习报告的解读方法是很值得的。通过几次实践，你就能浏览一下报告就能立即找出问题的所在了。

Mysqsla (MySQL Statement Log Analyzer) 是另一款有用的工具。你可以用它分析由服务器上所有查询生成的日志、慢速查询日志（慢速的意思是该查询所需要的时间超出了配置里的最大时间）、或者任何其他日志。它可以接纳多种日志格式，然后能即刻对它们进行分析。更多关于分析 MySQL 日志的内容，请查看第 65 页的“更精细地控制日志”。

该网站上的其他程序可以帮你分析一台服务器的索引使用情况，以及检查跟 MySQL 相关的网络数据流。

14.3.2 Maatkit 分析工具

Maatkit Analysis Tool

Maatkit 是 Baron Schwartz 的另一项创造，是一系列命令行工具的集合。所有工具都是用 Perl 编写的，用来补充 MySQL 未能提供的那些重要功能。你可以在 <http://maatkit.sourceforge.net> 上找到它们，包括了分析工具和其他一些小功能。

597 这些分析工具里的其中一个 `mk-query-profiler`，当它监测你的服务器状态变量时能够执行查询，并会打印出一个具体且易读的关于查询执行前后的系统状态变量差异情况的报告。这份报告能使你对你的查询能有更深的理解，而不仅限于在它执行的时候。

你可以将查询通过管道传入 `mk-query-profiler` 的标准输入口，指定一个或多个查询文件，或者仅仅是要求它监测你的服务器而不运行任何查询（当你在运行一个外部应用时，这种方式会用得着）。你也能让它运行 shell 命令而不是查询。

`mk-query-profiler` 的报告分成好几个段落。默认的情况下，这个报告打印出的是一批摘要信息，但是，你也可以得到一份包含了每一个查询或者所选择的查询的报告，然后加载到 `mk-profile-compact` 的辅助工具里方便地进行比较。

以下是这份报告的主要段落：

- “Overall stats” 段罗列了一些基本信息，例如执行时间、命令的数目和网络流量。
- “Table and index accesses” 段显示了这个批处理里有多少个不同类型的执行计划。如果你看到有很多表扫描，这可能意味着你没有合适的索引可用于查询。
- “Row operations” 段显示了底层处理器的数目和/或这个批处理产生的 InnoDB 操作的数目。糟糕的查询计划会引发更多的底层操作。
- “I/O operations” 段显示的是这个批处理引发了多少内存和磁盘数据流。有一个可比较的段是显示跟 InnoDB 相关的数据操作。

总而言之，这份报告可以细致地描述出服务器正在处理多少种类型的任务，这比仅仅是衡量查询所需要的时间要更有价值得多。例如，它可以帮你筛选这样两个查询：它们在低负载时，在一个小数据集上都用几乎相等的时间完成了查询；但是，当使用到的数据量变得很大的时候，或者说在高负载的时候，它们的表现就有了巨大的差异。它也可以被用来验证是否你的优化起到了作用，在这种应用环境下，它就像是一个微型的基准测试工具。

在这个工具箱里还有另外几种分析工具：

mk-visual-explain

重构从 EXPLAIN 里取得的查询执行计划，然后以更易懂的树状图来显示。当查询计划很复杂的时候，这个尤为有用。查看 EXPLAIN 的输入，我们看到的是数百行语句，这样的长度几乎没人能理解它们。mk-visual-explain 作为一个教学工具也很有用，或者当你试着去学习怎么看懂 EXPLAIN 输出的时候，它也能帮得上忙。

mk-duplicate-key-checker

它能识别出重复的或冗余的索引和外键——这会影响到性能。更多内容请查看第 127 页的“多余和重复索引”。

mk-deadlock-logger

监测 InnoDB 里的死锁并将它们记录在一个文件或表里。

mk-heartbeat

精确测量复制的延迟时间，无须检查 SHOW SLAVE STATUS（它也不总是正确）。默认情况下，它保留了过去 1 分钟、5 分钟和 15 分钟里的移动平均值。这是针对本书第一版里那个心跳（Heartbeat）脚本提到过的更完整更具可配置性的实现。

14.4 MySQL 的辅助工具

MySQL 辅助工具

MySQL 里有几个工具是为了消除 MySQL 提供的功能与它自带的命令行工具之间的隔阂。这一节里将讨论这些辅助工具。

14.4.1 MySQL Proxy

MySQL Proxy 项目是由 MySQL AB 开发并维护的，它以 GPL 许可证的形式发布，将来可能会随着 MySQL 服务器一起发布。在本书编写时，它诞生还不到一年时间，处于一个快速发展期里。（注 2）现在，你能在 <http://www.mysql.com> 的 Community 版块上找到它。它的文档已经集成在了 MySQL 使用手册里。

该软件的核心概念是一个带有状态的应用，它能解析 MySQL 的客户机/服务器协议，也能处于客户机与服务器之间，透明地转发它们的消息。一个客户端应用能够连接到它上面，跟连到服务器一样，这时，代理就会创建一个连接，连到一台真正的 MySQL 服务器——这种行为就像一个中间商。

这样一个单独的功能也可以被用于许多应用里（比如负载均衡和故障转移），但是代理更进了一步。既然它能解析客户端/服务器协议，那么它就能审查查询和响应结果。它还内建了一个 Lua 解释器，这样你就能编写自定义脚本，对查询和响应结果做任何你能想象到的操作，以下几种就是可能的应用方式：

- 重写或过滤查询。例如，你可以编写一个脚本识别出专门传递给代理的命令，然后做这些特定的操作，而不是把查询传递给服务器。
- 产生新的结果集，使它看上去像是来自 MySQL 服务器的，或者丢弃服务器生成的结果集。
- 基于它监测的内容，动态地调优 MySQL 服务器。例如代理能够开启或关闭那些缓慢的查询，或者能够跟踪查询统计信息，在响应一个查询时，显示出响应时间的柱状图。
- 在每次事务提交的时候注入查询语句（例如创建一个全局的事务标识）。

以上这些应用都有可工作的代码，你可以从在线文章和代码库里下载到。应用的可能性几乎是无限的。有创造力的用户会从中发掘出我们从没想到过的使用方法。如果你在考虑如何使用代理的时候有一些麻烦，我们建议你阅读几篇 Giuseppe Maxia 或 Jan Kneschke 写的文章。

14.4.2 Dormando 的 MySQL 代理

另一个使用 GPL 授权的代理项目是 Dormando 的 MySQL 代理，它几乎跟 MySQL Proxy 同时出现（事实上是它最早引入 Lua 脚本功能）。它是对 MySQL Proxy 项目的一个响应，到目前为止还没有发布，而且最终的许可证方式也没确定下来。就像 MySQL Proxy 一样，它的改变也很快，因此，你应该去检查一下它最新的版本，看看目前处于什么样的状态。它的网站是 <http://www.consoleninja.net/code/dpm/>。

14.4.3 Maatkit 辅助脚本

我们在前面罗列分析工具的时候已经提到过 Maatkit 了，其实，它还包括了许多辅助脚本。在这些脚本里，最重要的是 mk-table-checksum 和 mk-table-sync，在第 380 页的“确定主、从服务器是否一致”里，我们已经讲到过它们。除此之外，Maatkit 还包括了以下这些脚本：

注 2：MySQL Proxy 进展迅速，当你在阅读本书的时候，这些信息可能已经过时了。

mk-archiver

运行清除和归档任务，帮你清理表里不需要的数据。这个工具是被设计用来移动数据的，而无需动用 OLTP 查询。但是，你也可以用它来建立数据仓库，或者查找和移除陈旧的数据。它能把数据写到一个文件里，并且/或者写到任何 MySQL 实例上的另一个表里。它还有一个插件机制，使定制任务更加容易实现，举例来说，你可以使用一个插件，让它在系统往日志表里插入数据时，自动在数据仓库里生成一个摘要表。

mk-find

跟 Unix 的 find 命令类似，但是，它是用于 MySQL 的数据库和表。

mk-parallel-dump

执行多线程逻辑备份。为了能在多 CPU 或多磁盘的系统里更快速地备份，它能把每个表分割成所需要大小的几个分块。实际上，你可以用它把任何工具都封装成多线程的形式，因此，它在多线程 CHECK TABLE 或 OPTIMIZE TABLE 操作方面也很有用处（以此为例）。许多类型的任务都能因此在多 CPU 或多磁盘获取并行处理的好处。

mk-parallel-restore

它是 mk-parallel-dump 的伴随程序。它并行地把文件加载到 MySQL 里。这个工具能够通过 LOAD DATA INFILE 直接加载分界符文件，或者将 SQL 文件委托给 mysql 客户端程序导入。它对许多加载操作做了一个灵活的封装，例如可以通过命名管道加载压缩文件。

mk-show-grants

能对 GRANT 语句进行规整化、排除、分离和排序，使它更易于在命令行里操作。有一种很有趣的应用是你的数据库权限存储在一个版本控制系统里，不会有任何伪造的更改。

mk-slave-delay

使从服务器的更新滞后于它的主机，以便于灾难还原。如果有一条破坏性 SQL 语句在主机上执行后，你能在从服务器执行这条语句之前将它停止掉，重放二进制日志直到这条语句，然后将其提升为主机。这样的做法要快于在主机上重加载最近的一个备份，然后重新自备份以来的所有日志。

mk-slave-prefetch

实现了在第 402 页“为从服务器线程准备好缓存”里讨论过的技术。在某些工作负载下，它能使复制在从服务器上运行得更快。

mk-slave-restart

使从服务器在遇到一个错误后重新启动。

mk-table-checksum

在一台或多台服务器上并行地生成表内容的校验和，或者在各个复制里进行校验和查询，以检验你的从服务器上的数据是否一致。

mk-table-sync

更高效地找出表之间的差异，并生成用于解析它们的最小的 SQL 命令集。它也能在复制里面操作。

Baron 会经常加入新的工具，所以，这个列表可能也过时了。你可以在 <http://maatkit.sourceforge.net> 上获取到最新的工具和文档。

601 14.5 更多的信息来源

如果你正在 MySQL 上做着许多重复的或易出错的手工操作，而其他人可能已经创建了一个工具或脚本能减轻你的负担，问题在于你如何能找到这个工具。我们是通过阅读 Planet MySQL 博客聚合器 (<http://www.planetmysql.org>) 和 MySQL Forge 社区网站 (<http://forge.mysql.com>) 了解到许多我们喜爱的工具的。通常，这些网站都是学习 MySQL 的优秀资源。它们也有邮件列表、IRC 频道和论坛，在那里，你经常可以从那些友好的高手那里得到问题的解答（但是你要先搜索一下存档!）。

讨论会是另一个了解到 MySQL 的工具和技术的重要场所。即使你无法参加这样的讨论会，你也能经常性地下载到讲座的幻灯片或者在线观看会议的视频。

关于有些复杂工具（例如 Nagios）的更多信息，你也能找到专门讲解它们的书来看，在那些书里获得的资源将远远超过我们在本章里所提到的一点点相关内容。

大文件传输

Transferring Large Files

在管理 MySQL、初始化服务器、克隆从服务器和进行备份/还原操作时，复制、压缩和解压缩大文件（常常是跨网络的）是很常见的任务。能够最快最好完成这些任务的方法往往不总是显而易见的，而方法好坏的差异却非常显著。这个附录将讲述的是使用 Unix 的辅助功能，将一个大尺寸的备份文件从一台服务器复制到另外一台服务器的几种方法。

通常，我们使用的都是**未压缩**的文件，例如一台服务器上的 InnoDB 表空间和日志文件，当然，你也会想到在把文件复制到目的地之后再将它压缩一下。另外一个常见的场景是起初就使用了**压缩文件**，例如备份镜像文件，它就是一个压缩过的文件。

如果你的网络传输能力有限，那么用压缩格式在网络间发送文件是个不错的办法。你可能也需要一个安全的传输途径，使你的数据不会被损坏。这对于备份镜像文件来说，是一个很常见的需求。

A.1 复制文件

Copying Files

这个任务按照下面的步骤来完成会很有效率：

1. （可选的）压缩数据。
2. 发送到另外一台机器上。
3. 把数据解压缩到最终的目的地。
4. 在复制完成后，对文件进行校验，确认其没有被损坏。

我们对各种能达成这些目标的方法逐一进行了基准测试。本附录的余下部分就是向你展示我们怎么完成这个过程，以及我们找到的最快速的复制方法。

我们在本书里已经讨论过很多观点，例如备份，你可能想确认一下哪一台机器做压缩会更好一点。如果你有足够的网络带宽，你还是复制未压缩形式的备份镜像文件为好，这样在你的 MySQL 服务器上可以省出 CPU 资源给查询使用。

A.1.1 一个简单的示例

A Naïve Example

我们以一个简单的示例来开始：它安全地将一个未压缩的文件从一台机器发送到另外一台上，途中将它进行压

缩，然后再解压。在源服务器（我们称之为服务器 1）上，我们执行如下命令：

```
server1$ gzip -c /backup/mydb/mytable.MYD > mytable.MYD.gz
server1$ scp mytable.MYD.gz root@server2:/var/lib/mysql/mydb/
```

然后，在服务器 2 上：

```
server2$ gunzip /var/lib/mysql/mydb/mytable.MYD.gz
```

这大概就是最简单的实现方法了，但是，它不是非常有效率，因为这一系列步骤里包含了压缩、复制和解压缩，它们都需要读写磁盘，速度会很慢。上述命令的真正操作依次是这样的：gzip 在服务器 1 上既要读又要写；scp 在服务器 1 上读，在服务器 2 上写；gunzip 在服务器 2 上既要读又要写。

A.1.2 一步到位的方法

A One-Step Method

这个方法更有效率一些，它将压缩、复制文件和在传输的另一端解压缩文件全部放在一个步骤里完成。这一次我们使用了 SSH，SCP 就是基于这个安全协议的。下面是我们在服务器 1 上执行的命令：

```
server1$ gzip -c /backup/mydb/mytable.MYD | ssh root@server2
"gunzip -c - > /var/lib/mysql/mydb/mytable.MYD"
```

这个方法通常比第一个方法好，因为它极大地降低了磁盘 I/O：磁盘活动被降低到只要在服务器 1 上读，在服务器 2 上写。这也使得磁盘操作更加有序。

你也可以使用 SSH 内建的压缩来做这个，但是我们特意向你展示用管道来做压缩和解压缩，是因为它们能给予你更大的兼容性。例如，假如你不想在另一端解压缩文件，或者是不想使用 SSH 的压缩。

通过调节几个选项，你还能提高这个方法的效率，例如给 gzip 增加一个 -1 选项，使其压缩得更快。这个选项通常不会降低压缩率太多，但是，能明显提高压缩速度，这才是重点。你也可以使用不同的压缩算法。例如，如果你想获得很高的压缩率，不在乎它会花费多少时间，那么，你就可以使用 bzip2 来代替 gzip。如果你想让压缩速度非常快，你可以使用基于 LZO 的归档方法来代替 gzip，这样压缩后的数据会比通常的压缩文件大大约 20%，但是，压缩的速度快了约 5 倍。

8 A.1.3 避免加密方面的系统开销

Preventing the System Overhead

SSH 还不是跨网传输数据的最快方法，因为它增加了加解密系统开销。如果你不需要加密，那就使用 netcat 把“裸”数据进行跨网复制。你通过 nc 调用这个工具作为非交互式操作，这正是我们想让你做的。

这里有一个例子。首先，让我们在服务器 2 的 12345 端口（任何闲置的端口都可以）上监听文件的到来，并把任何发送到该端口的东西都解压缩到指定的数据文件里：

```
server2$ nc -l -p 12345 | gunzip -c - > /var/lib/mysql/mydb/mytable.MYD
```

在服务器 1 上，我们开启另一个 netcat 实例，设定发送的目的地是服务器 2 上的那个监听端口。-q 选项告诉 netcat 当到达输入文件的末尾时就关闭连接。这个操作会触发监听实例关闭正在接收的文件，并随之退出：

```
server1$ gzip -c - /var/lib/mysql/mydb/mytable.MYD | nc -q 1 server2 12345
```


有一个更方便的技术是使用 tar，这样，文件名称也会通过网络发送过去，以此消除另一个错误的来源，自动将文件写到它们的正确位置上。z 选项告诉 tar 使用 gzip 做压缩和解压缩。下面是在服务器 2 上执行的命令：

```
server2$ nc -l -p 12345 | tar xvfz -
```

以下是在服务器 1 上执行的命令：

```
server1$ tar cvzf - /var/lib/mysql/mydb/mytable.MYD | nc -q 1 server2 12345
```

你可以把这些命令装配到一个单独的本里，它会高效地压缩、复制大量的文件到网络连接里，然后在另一端将它们解压缩。

A.1.4 其他选项

Other Options

另外一个选项是 rsync。rsync 非常简便，因为它易于在源和目标之间做镜像，还可以断点续传。但是，当它的二进制差异算法无法很好地发挥作用时，它的功能也会显得大打折扣。你可以考虑将它用在类似这种情形之下：你知道文件中的大部分内容都不需要传输——例如，完成一个中途退出的 nc 复制操作。

当你还没处于危急关头时，你应该针对文件传输做一些实验，因为具体哪一种方法最快取决于你的系统，只有通过反复的尝试，才能找出最快的传输方法。其中，最大的影响因素是服务器上的磁盘驱动器、网卡和 CPU 的个数，以及它们之间相对的最大速度。有个不错的方法是监控 vmstat -n 5，看看磁盘或 CPU 是否就是速度的瓶颈。

如果你有闲置的 CPU，就可能通过运行并行复制操作来加快整个过程。相反，如果 CPU 就是这个过程的瓶颈，而磁盘和网络的承载能力还比较充裕，那就略过压缩这个步骤。在处置导出和还原时，出于速度的考虑，并行地执行这些操作往往是个不错的主意。此外，监控你的服务器性能看看它是否还有闲置的承载能力。过度的使用并行处理反而会降低处理速度。

606

A.2 文件复制的基准测试

File Copy Benchmark

为了便于比较，表 A-1 显示的是在局域网里通过一块标准的百兆以太网卡复制一个样本文件能达到的最快速度。这个文件未压缩时的大小是 738MB，使用 gzip 默认选项压缩后是 100MB。源和目的机器都有充足的可用内存、CPU 资源和磁盘空间，仅网络是其中的瓶颈。

表 A-1：跨网复制文件的基准测试

| 方法 | 时间（秒） |
|-----------------------------|------------------|
| rsync without compression | 71 |
| scp without compression | 68 |
| nc without compression | 67 |
| rsync with compression (-z) | 63 |
| gzip, scp, and gunzip | 60 (44 + 10 + 6) |
| ssh with compression | 44 |
| nc with compression | 42 |

请注意当跨网发送文件时，压缩过的文件能起的作用有多大——其中最慢的 3 个方法都没有压缩文件，然而，它们的速度还是有所不同。如果你拥有较慢的 CPU 和磁盘，但有一个千兆以太网连接，那么，读取和压缩数据可能会成为瓶颈，跳过压缩这个步骤会更快一些。

另外，使用快速压缩，例如 `gzip-fast`，要比那些使用默认压缩选项的压缩方法快一点点，后者使用了大量的 CPU 时间去压缩文件。

传输数据的最后一个步骤就是对传输过来的副本进行校验，确认其没有损坏。你可以使用各种方法来做这个事情，例如 `md5sum`，但是它需要对文件作一次全扫描，系统开销会比较大。这也是为什么压缩很有用处的原因之一：每一个压缩文件一般都至少包含了一个循环冗余检验码 (CRC)，它能捕捉到文件数据里的任何错误。这样，你就可免费得到错误检查的功能。

使用 EXPLAIN

Using EXPLAIN

本附录将为你展示如何通过调用 EXPLAIN 来获取查询执行计划的信息，以及如何解读输出结果。EXPLAIN 命令是找出查询优化器如何执行查询的主要方法。尽管这项功能特性有许多局限性，也不总是“吐露实情”，但是，它的输出仍然是可供使用的最好信息，值得你学习，这样你就可以对你的查询执行情况作一个有根据的推测。

B.1 调用 EXPLAIN

invoking EXPLAIN

要使用 EXPLAIN，只需把 EXPLAIN 这个单词放在查询语句的关键字 SELECT 前面就可以了。MySQL 会在查询里设置一个标记。当它执行查询时，这个标记会促使 MySQL 返回执行计划里每一步的信息，用不着真正执行它。它会返回一行或多行，每行都会显示执行计划的每一个组成部分，以及执行的次序。

以下是 EXPLAIN 可能输出结果里最简单的一种：

```
mysql> EXPLAIN SELECT 1\G
***** 1. row *****
      id: 1
    select_type: SIMPLE
        table: NULL
         type: NULL
possible_keys: NULL
         key: NULL
        key_len: NULL
         ref: NULL
         rows: NULL
      Extra: No tables used
```

查询里的每一张表对应输出结果里的一行。如果你的查询联接了两张表，这里就会有两行输出。一个别名表也会作为独立的一张表来计算，因此，如果你把一个表和自己联接起来，输出结果里也会有两行。在这里，“表”的含义变得很广泛：它可以意味着是一个子查询、一个 UNION 结果等等。在下文里，你会看到为什么会变成这个样子。

EXPLAIN 有两种重要的变体：

- EXPLAIN EXTENDED 的行为特点跟普通的 EXPLAIN 很相似，但是，它告知服务器把执行计划“反编译”成 SELECT 语句，然后你立即运行 SHOW WARNINGS 就能看到这些生成的语句。这些语句是直接来自于执行计划，不是原始的 SQL 语句，因此，会精简为一个数据结构，在大多数情况下跟原始语句不一样。通过查看这些语句，你可以知道查询优化器到底是怎么转换你的查询语句的。EXPLAIN 在 MySQL 5.0 及更新的版本里都可以使用，在 MySQL 5.1 里它又增加了一个额外的过滤列（下文中会讲到）。

- EXPLAIN PARTITIONS 显示的是查询要访问的数据分片——如果有分片的话。它只能在 MySQL 5.1 及更新的版本里使用。关于数据分片的更多细节，请查看第 257 页的“分区表”。

有一个常见的误解就是认为当你在查询里添加 EXPLAIN 后，MySQL 不会真正地执行它。事实上，如果在 FROM 子句里包含了一个子查询的话，MySQL 还是会执行这个子查询的，并把查询结果放入一个临时表里，然后完成对外层查询的优化工作。在它完成外层查询的全部优化工作之前，它就是这样处理所有子查询，这是针对 EXPLAIN 必须要做的事情。这意味着如果查询语句里包含了使用 TEMPTABLE 算法的开销昂贵的子查询或者视图时，EXPLAIN 也能引发服务器大量的处理任务。

要记住 EXPLAIN 只是一个近似，没有更多的细节。有时，它是一个很好的近似，但是有时，它会远离真实情况。以下就是它的几个局限性：

- EXPLAIN 不会告诉你任何关于触发器、存储函数或 UDF 对你查询的影响情况。
- 对于存储过程，它是没有用的，但是你可以手工地分离出所有组成的查询语句，然后用 EXPLAIN 一条条单独执行。
- 它不会告诉你 MySQL 在查询执行时所做的那些优化工作。
- 一些显示出来的统计信息都是估算的，不是很精确。
- 它无法显示出一个查询执行计划里的所有信息。(MySQL 的开发者正尽可能地往里面添加更多的信息。)
- 它无法区分同名的一些东西。例如，它使用“filesort”作为内存排序和临时文件，用“Using temporary”来显示磁盘上的临时文件和内存里的临时文件。
- 它会误导你。举例来说，它会把一个带有小范围 LIMIT 的查询显示为全索引扫描。(MySQL 5.1 的 EXPLAIN 能够更精确地显示要检查到的行数，但是，早期版本没有把 LIMIT 考虑进去。)

B.1.1 重写非 SELECT 查询

Rewriting Non-SELECT Queries

MySQL 只能解释 SELECT 查询，无法解释存储程序的调用、INSERT、DELETE 和其他类似的语句。然而，你可以重写一些非 SELECT 查询，使它们能够被 EXPLAIN。为了做到这一点，你只需要把语句转换为对等的访问同样那些列的 SELECT 就行了。所有提及的列都必须出现在 SELECT 列表里、联接子句里或者 WHERE 子句里。

举例来说，假设你要重写下面这个 UPDATE 语句，使它能被 EXPLAIN：

```
UPDATE sakila.actor
  INNER JOIN sakila.film_actor USING (actor_id)
SET actor.last_update=film_actor.last_update;
```

下面这个 EXPLAIN 语句不等于那个 UPDATE 语句，因为它没有要求服务器为每一个表获取 last_update 列：

```
mysql> EXPLAIN SELECT film_actor.actor_id
-> FROM sakila.actor
-> INNER JOIN sakila.film_actor USING (actor_id)\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: actor
```

```

        type: index
possible_keys: PRIMARY
        key: PRIMARY
        key_len: 2
        ref: NULL
        rows: 200
        Extra: Using index
***** 2. row *****
        id: 1
        select_type: SIMPLE
        table: film_actor
        type: ref
possible_keys: PRIMARY
        key: PRIMARY
        key_len: 2
        ref: sakila.actor.actor_id
        rows: 13
        Extra: Using index

```

这个差别非常重要。输出结果显示 MySQL 将会使用覆盖索引（Covering Index）。例如，在获取和更新 `last_updated` 列时无法使用到它。下面这段语句更接近原始的样子：

```

mysql> EXPLAIN SELECT film_actor.last_update, actor.last_update
-> FROM sakila.actor
-> INNER JOIN sakila.film_actor USING (actor_id)\G
***** 1. row *****
        id: 1
        select_type: SIMPLE
        table: actor
        type: ALL
possible_keys: PRIMARY
        key: NULL
        key_len: NULL
        ref: NULL
        rows: 200
        Extra:
***** 2. row *****
        id: 1
        select_type: SIMPLE
        table: film_actor
        type: ref
possible_keys: PRIMARY
        key: PRIMARY
        key_len: 2
        ref: sakila.actor.actor_id
        rows: 13
        Extra:

```

像这样重写查询不能算是一种精密科学，但是，它通常已经能足够帮你理解查询将要做的事情。

领会这一点很重要：没有真正“对等”的读查询可以把一个写查询显示给你看。一个 `SELECT` 查询只需要找到数据的一个副本，然后返回给你；而任何更新数据的查询必须在所有索引里找出并更新全部符合条件的副本，它往往要比对等的 `SELECT` 查询多产生很多的系统开销。

B.2 EXPLAIN 里的列

Too Columns in EXPLAIN

EXPLAIN 的输出总是有着相同的列(除了 MySQL 5.1 里的 EXPLAIN EXTENDED 增加了一个过滤列,以及 EXPLAIN PARTITIONS 增加了一个分片列)。其中变化的是行的数目和内容。但是,为了让我们的示例保持清晰,我们在这个附录里不会总是显示出所有的列。

在下一个小节里,我们将为你讲述 EXPLAIN 输出结果里每一个列的含义。要记住行的出现次序就是 MySQL 实际执行它们的次序,它们跟原始 SQL 语句的次序不总是相同的。

B.2.1 id 列

The id Column

id 列总是包含一组数字,根据它可识别出 SELECT 是属于哪一行的。如果语句里没有子查询或者联接,那整个输出里就只有一个 SELECT,这样一来,每一行在这个列上都会显示一个 1。另外,内层的 SELECT 语句通常按它们在原始语句的位置顺序来编号。

611 MySQL 把 SELECT 查询分成简单和复杂两种类型,复杂类型又可以分成三个宽泛的大类:简单子查询、所谓的衍生表(子查询在 FROM 子句里)(注 1)和 UNION。这里有一个简单的子查询:

```
mysql> EXPLAIN SELECT (SELECT 1 FROM sakila.actor LIMIT 1) FROM sakila.film;
+----+-----+-----+...
| id | select_type | table |...
+----+-----+-----+...
| 1  | PRIMARY    | film  |...
| 2  | SUBQUERY   | actor |...
```

在 FROM 子句和 UNION 里的子查询会给 id 列带来更大的复杂性。这里有一个基本的子查询位于 FROM 子句里的例子:

```
mysql> EXPLAIN SELECT film_id FROM (SELECT film_id FROM sakila.film) AS der;
+----+-----+-----+...
| id | select_type | table      |...
+----+-----+-----+...
| 1  | PRIMARY    | <derived2> |...
| 2  | DERIVED    | film       |...
```

正如你所知道的那样,这个查询执行时会使用到临时表。MySQL 内部会通过外层查询里的一个别名(der)来引用这个临时表,这个现象你可以在复杂查询的 ref 列里看到。

最后是一个 UNION 查询:

```
mysql> EXPLAIN SELECT 1 UNION ALL SELECT 1;
+----+-----+-----+...
| id | select_type | table      |...
+----+-----+-----+...
```

注 1: 语句“FROM 子句里的子查询是一个衍生表”是对的,但是,“一个衍生表就是 FROM 子句里的子查询”是错的。其中的术语“衍生表”在 SQL 里有着更宽泛的含义。

| | | | |
|------|--------------|------------|-----|
| 1 | PRIMARY | NULL | ... |
| 2 | UNION | NULL | ... |
| NULL | UNION RESULT | <union1,2> | ... |



要注意 UNION 输出结果里有额外的一行。UNION 的结果总是放在一个临时表里的，MySQL 就从这个临时表里读出结果。临时表不会在原始 SQL 语句里出现，因此它的 id 列是 NULL。跟前面那个例子（说明子查询在 FROM 子句里的例子）相反的是：该查询产生的临时表是放在最后一行显示，而不是第一行。

到目前为止，一切都非常直接易懂，但是，混合了这三种类型查询的语句会使输出变得更加复杂。在下文里，我们会看到一点。

B.2.2 select_type 列

The select_type Column

612

这个列显示了该行是简单 SELECT 还是复杂 SELECT（如果是后者，显示的就是它属于哪一种复杂类型）。值 SIMPLE 意味着这个查询里没有子查询或者 UNION。如果查询里有了任何复杂的子部分，最外层部分就被标记为 PRIMARY，其余部分会用以下几种标记类型：

SUBQUERY

在 SELECT 的目标里包含了一个子查询（换句话说就是子查询没在 FROM 子句里），那就标记为 SUBQUERY。

DERIVED

值 DERIVED 用于 FROM 子句里有子查询的情况。MySQL 会递归执行这些子查询，把结果放在临时表里。在内部，服务器就把它当作一个“衍生表”那样来引用，因为临时表就是源自子查询。

UNION

如果第 2 个以及之后的 SELECT 出现在 UNION 里，那就标记为 UNION。第 1 个 SELECT 标记为它就像外层查询的一部分那样被执行。这就是为什么前一个例子当中第 1 个 SELECT 作为 PRIMARY 出现在 UNION 里。如果 UNION 包含在一个 FROM 子句的子查询里，那么，它的第 1 个 SELECT 将被标记为 DERIVED。

UNION RESULT

从 UNION 临时表获取结果的 SELECT 被标记为 UNION RESULT。

除了这些值外，SUBQUERY 和 UNION 也能作为 DEPENDENT 和 UNCACHEABLE、DEPENDENT 来标记，这意味着 SELECT 依赖于外层查询里找到的数据。UNCACHEABLE 意味着 SELECT 里有某些东西阻止了 Item_cache 的缓存。（Item_cache 没有写在文档里，它跟查询缓存不是一回事，尽管它会因一些同样类型的结构而失效，例如 RAND() 函数。）

B.2.3 table 列

The table Column

这一列显示了当前行访问的是哪个表。在多数情况下，它的意思很清楚：这是表，或者 SQL 指定的一个别名表。

你可以从上到下读这一列，看看 MySQL 联接优化器为这个查询生成的联接次序。举例来说，在下面这个查询里，你能看到 MySQL 为它选择了一个完全不同的联接次序：


```
mysql> EXPLAIN SELECT film.film_id
-> FROM sakila.film
-> INNER JOIN sakila.film_actor USING(film_id)
-> INNER JOIN sakila.actor USING(actor_id);
+-----+-----+-----+...
| id | select_type | table | ...
+-----+-----+-----+...
| 1 | SIMPLE | actor | ...
| 1 | SIMPLE | film_actor | ...
| 1 | SIMPLE | film | ...
+-----+-----+-----+...
```

记得我们在第 127 页里，用左深树（Left-deep tree）的形式显示过“执行计划”。MySQL 的查询执行计划总是一个左深树。如果你把计划反转 to 另一边，你就能按次序读出那个叶子节点了，它们直接对应于 EXPLAIN 里的行。前面那个查询的计划看上去就是图 B-1。

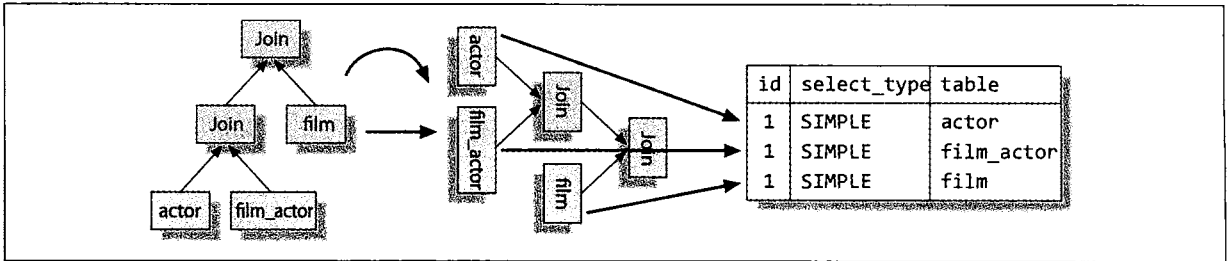


图 B-1：查询执行计划是如何对应 EXPLAIN 的行的

衍生表和联合

当 FROM 子句里有子查询或者有 UNION 时，表的列会变得更加复杂。在这种情况下，它引用的不是真正的“表”，因为 MySQL 创建的临时表只在查询执行的时候存在。

当有一个子查询在 FROM 子句的时候，表的列就变成 <derivedN> 的形式，这里的 N 就是子查询的 id。有一个“前向引用（Forward Reference）”总是存在的——换句话说，N 引用的是 EXPLAIN 输出里的下一行。

当有 UNION 时，UNION RESULT 的列里面就会包含一个 id 列表，这些 id 都出现在 UNION 里。这里总会有一个“后向引用（Backward Reference）”，因为 UNION RESULT 是紧跟着 UNION 里的那些行之后出现的。如果列表里有超过 20 个 id，那 table 列就会被删减，以免太长，而你也就看到所有的值了。幸运的是，你仍然可以推算出有哪些行被包括进去了，因为你能看到第一行的 id。所有来自该行和 UNION RESULT 之间的任何东西都会通过某种方式被包括进来。

一个复杂 SELECT 类型的示例

这里我们就把一个无意义的查询用作演示几种复杂 SELECT 类型的简洁示例：

```
1  EXPLAIN
2  SELECT actor_id,
3  (SELECT 1 FROM sakila.film_actor WHERE film_actor.actor_id =
4  der_1.actor_id LIMIT 1)
5  FROM (
6  SELECT actor_id
7  FROM sakila.actor LIMIT 5
8  ) AS der_1
9  UNION ALL
```

```

10 SELECT film_id,
11     (SELECT @var1 FROM sakila.rental LIMIT 1)
12 FROM (
13     SELECT film_id,
14         (SELECT 1 FROM sakila.store LIMIT 1)
15     FROM sakila.film LIMIT 5
16 ) AS der_2;

```

LIMIT 子句仅仅是为了图个方便，万一你打算不带 EXPLAIN 直接执行查询并查看结果了呢。以下是 EXPLAIN 的输出结果：

| id | select_type | table | ... |
|------|----------------------|------------|-----|
| 1 | PRIMARY | <derived3> | ... |
| 3 | DERIVED | actor | ... |
| 2 | DEPENDENT SUBQUERY | film_actor | ... |
| 4 | UNION | <derived6> | ... |
| 6 | DERIVED | film | ... |
| 7 | SUBQUERY | store | ... |
| 5 | UNCACHEABLE SUBQUERY | rental | ... |
| NULL | UNION RESULT | <union1,4> | ... |

我们特意构造出查询的每一部分会访问到不同的表，这样，你就能看到接下去会发生什么，但是，它仍然难以说清！我们就从头往下来讲解：

- 第 1 行就是对 der_1 的一个前向引用，该查询已被标记为<derived3>。它来自于原始 SQL 的第 2 行。想知道输出结果里的哪一行引用了<derived3>里的 SELECT 语句，请继续往下看……
- 在第 2 行，它的 id 是 3。之所以是 3 的原因是它是查询里第三个 SELECT 的一部分。又因为它嵌套在 FROM 子句的子查询里，它就被归为 DERIVED 类型。它来自于原始 SQL 的第 6、7 行。
- 第 3 行的 id 是 2。它来自于原始 SQL 的第 3 行。要注意到它紧跟在更大 id 的行之后，这暗示着它是在那一行之后执行的，这很有意义。它被列为 DEPENDENT SUBQUERY，这意味着它的结果依赖于一个外层查询（也就是一个相关子查询）的结果。在本例中，外层查询就是从第 2 行开始的 SELECT，数据是从 der_1 获取到的。
- 第 4 行是作为 UNION 列出的，这意味着它是 UNION 里第二个或之后的 SELECT。它的表是<derived6>，说明它的数据是从 FROM 子句的一个子查询里获取到的，然后附加到 UNION 的一个临时表里。如前所述，要找到能显示子查询的查询计划的 EXPLAIN 行，你还得往前看。
- 第 5 行是 der_2 子查询，对应于原始 SQL 的第 13、14、15 行。EXPLAIN 将它作为<derived6>来引用。
- 第 6 行是<derived6>的 SELECT 列表里的一个普通的子查询。它的 id 是 7，这非常重要……
- ……因为它大于 5，这是第 7 行的 id。为什么说它很重要呢？这是因为它显示了<derived6>子查询的边界。当 EXPLAIN 输出 SELECT 类型为 DERIVED 的一行时，它表明这是一个“嵌套范围”的开始，如果后续行的 id 更小的话（在本例中，5 比 6 小），那就说明嵌套范围已被关闭。这就让你知道第 7 行是 SELECT 列表的一部分，它从<derived6>那里取得了数据——也就是说，它是第 4 行的 SELECT 列表的一部分（对应于原始 SQL 的第 11 行）。这个例子非常易于理解，无须知道嵌套范围的意义和规则，但是，有时它就没有这么容易被理解。这一行里还有一个值得注意的事情是因为它属于用户变量，所以输出结果被列为 UNCACHEABLE SUBQUERY。

615

- 最后一行是 UNION RESULT。它代表着从 UNION 的临时表里读取行的进行阶段。你可以从这一行开始，回退到任何你希望的那一步。它会返回行的结果——它们的 id 是 1 和 4，依次引用了<derived3>和<derived6>。

正如你所看到的那样，把这些复杂 SELECT 类型结合起来，会导致 EXPLAIN 的输出结果非常难以读懂。了解其中的规则，会使它简单一点，但是除了反复实践，没有其他更好的替代办法了。

阅读 EXPLAIN 的输出结果常常会让你在列表里前后跳读。例如，再看一遍输出的第一行，仅仅盯着它看，是不会知道它是一个 UNION 的一部分的，只有当你看到输出的最后一行时才能明白过来。

B.2.4 type 列

The type Column

MySQL 的使用手册里说这一列显示的是“联接的类型”，但是，我们认为更确切的说法应该是“访问类型”——换句话说就是 MySQL 在表里找到所需行的方式。下面就是最重要的几种访问类型，从最差到最好：

ALL

这就是人们所称的全表扫描，通常意味着 MySQL 必须扫描整张表，从头到尾，去找到需要的行。（这里也有个例外，例如在查询里使用了 LIMIT，或者在 Extra 列里的显示使用了 distinct 或 notexists 等限定词。）

index

这个跟全表扫描一样，只是 MySQL 扫描表时按索引次序进行而不是行。它的主要优点是避免了排序；最大的缺点是要承担按索引次序读取整张表的开销。这通常意味着若是按随机次序访问行，开销将会非常大。如果你也看到过在 Extra 列里“使用索引”，这说明 MySQL 正在使用一个覆盖索引（请看第 3 章），它只扫描索引的数据，而不是按索引次序的每一行。在开销方面要比按索引次序的全表扫描少很多。

range

范围扫描就是一个有限制的索引扫描，它开始于索引里的某一点，返回匹配那个值域的行。这比全索引扫描好一些，因为它用不着遍历全部索引。常见的范围扫描就是在 WHERE 子句里带有 BETWEEN 或 > 的查询。

当 MySQL 使用一个索引去查找一系列值时，例如 IN() 和 OR 列表，它也会显示为范围扫描。然而，这两者其实是相当不同的访问类型，在性能上有重要的差异。更多信息可以查看第 134 页“什么是范围条件”。

此类扫描的开销跟索引类型相关。

ref

这是一种索引访问（有时也叫做索引查找），它返回所有匹配某个单独值的行。然而，它可能会找到多个符合条件的行，因此，它是查找和扫描的混合体。此类索引访问只有当使用一个非唯一性索引或者唯一性索引的非唯一性前缀时才会发生。把它叫做 ref 是因为索引要跟某个参考值相比较。这个参考值或者是一个常数，或者是来自前一个表里的多表查询的结果值。

ref_or_null 是 ref 之上的一个变体，它意味着 MySQL 必须进行二次查找，在初次查找的结果里找出 NULL 条目。

eq_ref

使用这种索引查找，MySQL 最多只返回一条符合条件的记录。这种访问方法你会在 MySQL 使用主键或者唯一性索引查找时看到，它会将它们与某个参考值作比较。MySQL 对于这类访问类型的优化做得非常好，

因为它知道它无须估计匹配行的范围，也不用在找到匹配行后再继续查找。

```
const, system
```

当 MySQL 能对查询的某部分进行优化，并转换成一个常量时，它就会使用这些访问类型。举例来说，如果你选择了某一行的主键放入 WHERE 子句里，MySQL 就能把这个查询转换为一个常量。然后就可以移除表的联接，更加有效地执行。

```
NULL
```

这种访问方式意味着 MySQL 能在优化过程中分解查询语句，在执行环节里，甚至用不着再访问表或者索引。举例来说，从一个索引列里选取最小值可以通过单独查找索引来完成，不需要在执行时访问表。

B.2.5 possible_keys 列

It's possible_keys Column

这一列显示了基于查询能够访问的列和选用的比较操作类型来决定哪个索引可以被用作查询。这个列表是在优化过程的早期被创建的，因此，有些罗列出来的索引可能对于后续优化过程是没用的。

B.2.6 key 列

It's Key Column

这一列显示了 MySQL 采用了哪一个索引来优化对该表的访问。如果该索引没有出现在 possible_keys 列，MySQL 选用它是出于另外一个原因——例如，它可能选择了一个覆盖索引，哪怕没有 WHERE 子句。

换句话说，possible_keys 揭示了哪一个索引能有助于行查找更加有效，而 key 显示的是优化器采用哪一个索引可以最小化查询成本（更多关于优化器的成本衡量值可以查看第 164 页的“查询优化过程”）。以下就是一个例子：

```
mysql> EXPLAIN SELECT actor_id, film_id FROM sakila.film_actor\G
***** 1. row *****
      id: 1
  select_type: SIMPLE
        table: film_actor
         type: index
possible_keys: NULL
         key: idx_fk_film_id
      key_len: 2
         ref: NULL
        rows: 5143
      Extra: Using index
```

B.2.7 key_len 列

It's Key_len Column

该列显示了 MySQL 在索引里使用的字节数。如果 MySQL 正在使用的是索引列里的某一个，那你就可以用这个值来算出具体是哪一列。要记住 MySQL 只能使用索引的最左边前缀。举例来说，sakila.film_actor 的主键是两个 SMALLINT 类型的列，一个 SMALLINT 有两个字节，那么索引里的每一项就是 4 个字节。以下就是一个查询的示例：

```
mysql> EXPLAIN SELECT actor_id, film_id FROM sakila.film_actor WHERE actor_id=4;
...+-----+-----+-----+-----+...
...| type | possible_keys | key | key_len | ...
...+-----+-----+-----+-----+...
...| ref | PRIMARY | PRIMARY | 2 | ...
...+-----+-----+-----+-----+...
```

基于查询结果里的 `key_len` 列，你能够推断出查询执行索引查找时仅仅使用了第一列，即 `actor_id`。当计算列的使用情况时，要确保考虑到了字符列里的字符集类型：

```
mysql> CREATE TABLE t (
->   a char(3) NOT NULL,
->   b int(11) NOT NULL,
->   c char(1) NOT NULL,
->   PRIMARY KEY (a,b,c)
-> ) ENGINE=MyISAM DEFAULT CHARSET=utf8 ;
mysql> INSERT INTO t(a, b, c)
->   SELECT DISTINCT LEFT(TABLE_SCHEMA, 3), ORD(TABLE_NAME),
->   LEFT(COLUMN_NAME, 1)
->   FROM INFORMATION_SCHEMA.COLUMNS;
mysql> EXPLAIN SELECT a FROM t WHERE a='sak' AND b = 112;
...+-----+-----+-----+-----+...
...| type | possible_keys | key | key_len | ...
...+-----+-----+-----+-----+...
...| ref | PRIMARY | PRIMARY | 13 | ...
...+-----+-----+-----+-----+...
```

在本次查询里的 13 个字节长度就是 `a` 列和 `b` 列的长度之和。`a` 列是 3 个字符，是 `utf8` 格式，所以每一个是 3 个字节；`b` 列是 4 个字节的证书。

MySQL 并不总是能显示出一个索引的真正被使用的次数。举例来说，如果你使用前缀模式匹配的 `LIKE` 查找，它就会显示出所使用列的全宽度。

`key_len` 列显示的是索引字段的最大可能长度，不是表内数据使用的真正字节数。在前一例中，MySQL 总是显示 13 个字节，哪怕是 `a` 列没有数据，连一个字符都没有。换句话说，`key_len` 是根据表的定义计算出来的，而不是通过表内数据检索出的。

B.2.8 ref 列

The ref Column

这一列显示了哪些来自上述表里的列或者常量正在被用于查找 `key` 列上的值。以下就是一个示例，其中使用了联接条件和别名的结合。要注意到 `ref` 列反映了 `film` 表是怎样以 `f` 为别名被用于查询语句里的：

```
mysql> EXPLAIN
-> SELECT STRAIGHT JOIN f.film_id
-> FROM sakila.film AS f
->   INNER JOIN sakila.film_actor AS fa
->     ON f.film_id=fa.film_id AND fa.actor_id = 1
->   INNER JOIN sakila.actor AS a USING(actor_id);
...+-----+-----+-----+-----+...
...| table | ... | key | key_len | ref | ...
...+-----+-----+-----+-----+...
...| a | ... | PRIMARY | 2 | const | ...
...| f | ... | idx_fk_language_id | 1 | NULL | ...
...| fa | ... | PRIMARY | 4 | const,sakila.f.film_id | ...
...+-----+-----+-----+-----+...
```

B.2.9 rows 列

The rows Column

这一列显示的是 MySQL 估计的为了找到所需的行而要读取的行数。这个数字是内嵌循环联接计划里的循环数目。也就是说它不是 MySQL 认为它最终要从表里读取出来的行数，而是 MySQL 为了找到符合查询的每一点上标准的那些行而必须进行读取的行的平均数。（这个标准包括 SQL 里给定的条件，以及来自联接次序上前一个表的当前列。）

根据表的统计信息和索引的选用情况，这个估算可以很精确。在 MySQL 5.0 及更早的版本里，它也不能反映出 LIMIT 子句。举例来说，下面这个查询不会真的检查 1 022 行：

```
mysql> EXPLAIN SELECT * FROM sakila.film LIMIT 1\G
...
      rows: 1022
```

通过把所有 rows 列的值都乘起来，你可以粗略地估算出整个查询会检查的行数。例如，以下这个查询大约会检查 2 600 行：

```
mysql> EXPLAIN
-> SELECT f.film_id
-> FROM sakila.film AS f
->   INNER JOIN sakila.film_actor AS fa USING(film_id)
->   INNER JOIN sakila.actor AS a USING(actor_id);
...+-----+...
...| rows |...
...+-----+...
...|  200 |...
...|   13 |...
...|    1 |...
...+-----+...
```

要记住这个数字是 MySQL 认为它要检查的行数，不是结果集里的行数。同时也要认识到有很多优化手段，例如联接缓冲区和缓存，都无法影响到行数的显示。MySQL 可能不必真的读入所有它估计到的行，它也不知道任何关于操作系统或硬件缓存的信息。

620

B.2.10 filtered 列

The filtered Column

这一列是在 MySQL 5.1 里新加进去的，当你使用 EXPLAIN EXTENDED 时，才会出现。它显示的是针对表里符合某个条件（WHERE 子句或联接条件）的记录数的百分比所作的一个悲观估算。如果你把 rows 列和这个百分比相乘，你就能看到 MySQL 估算的它将和查询计划里前一个表联接的行数。在本书写作的时候，优化器只有在在使用 ALL、index、range 和 index_merge 访问方法时才会用估算。

为了说明这一列的输出形式，我们创建了下面这样一张表：

```
CREATE TABLE t1 (
  id INT NOT NULL AUTO_INCREMENT,
  filler char(200),
  PRIMARY KEY(id)
);
```

然后，我们往表里插入 1 000 行记录，并在 filler 列里随机填充一些文字。它的用途是防止 MySQL 在我们将

要运行的查询里使用覆盖索引：

```
mysql> EXPLAIN EXTENDED SELECT * FROM t1 WHERE id < 500\G
***** 1. row *****
      id: 1
    select_type: SIMPLE
        table: t1
         type: ALL
possible_keys: PRIMARY
         key: NULL
        key_len: NULL
         ref:  NULL
         rows: 1000
    filtered: 49.40
      Extra: Using where
```

MySQL 可以使用范围访问从表里获取到所有 ID 不超过 500 的行，但是，它没这么做，这是因为它还能去除大约一半的记录，它认为一个全表扫描也不是太昂贵。因此，它使用了一个全表扫描和一个 WHERE 子句来过滤输出。它知道使用 WHERE 子句可以从结果里过滤掉多少条记录，因为范围访问的成本是可以估算出来的。这也就是为什么 49.40% 会出现在 filtered 列上的原因。

621 B.2.11 Extra 列 The Extra Column

这一列包含的是不适合在其他列显示的额外信息。MySQL 帮助手册里记录的大多数值都会在这里出现。在本书里，我们已经提到过其中的许多。

以下这些是你常见的最重要的值：

“Using index”

该值表示 MySQL 将使用覆盖索引，以避免访问表（请查看第 120 页的“覆盖索引”）。不要把覆盖索引和索引访问类型弄混了。

“Using where”

这意味着 MySQL 服务器将在存储引擎收到行后进行后-过滤（Post-filter）。许多 WHERE 条件里包含的属于索引的列，当它（如果它）读取索引的时候，就能被存储引擎检验，因此，不是所有带 WHERE 子句的查询会显示 “Using where”。有时 “Using where” 的出现就是一个提示：查询可受益于不同的索引。

“Using temporary”

这意味着 MySQL 在对查询结果排序时会使用一张临时表。

“Using filesort”

这意味着 MySQL 会对结果使用一个外部索引排序，不是从表里按索引次序来读取行。MySQL 有两种文件排序算法，在 300 页的 “Optimizing for filesorts” 里你可以读到相关内容。任何一种方式都可以在内存或磁盘上进行。EXPLAIN 无法告诉你 MySQL 将使用的是哪一种文件排序，也不会告诉你排序会在内存里还是磁盘上进行。

“range checked for each record (index map: N)”

这个值意味着没有好的索引可用，新的索引将在联接的每一行上被重新估算。*N* 是显示在 `possible_keys` 列索引的位图，这是一个冗余。

B.3 可视化 EXPLAIN

Visual EXPLAIN

MySQL 的开发者已经说过了他们更喜欢把 EXPLAIN 的输出格式化成一棵树，这是执行计划更精确的一种表示。MySQL 用户已经在盼望能有更进一步的提高。实际上，EXPLAIN 查看查询计划的方式确实有点笨拙，树状结构也不适合表格化信息的输出。当 `Extra` 列里有大量的值时，笨拙的显示问题就会显得更加突出；使用 UNION 时也是这样。UNION 跟 MySQL 能做的其他类型的联接不太一样，它跟 EXPLAIN 不太合得来。

如果对 EXPLAIN 的规则和特性有了充分的了解，使用树形结构的执行计划也是很方便的，但是，这总归有点枯燥，最好还是有一款自动的工具可用。Maatkit（请查看第 14 章）包含了一个 `mk-visual-explain`，它就是这样 622 一个工具。

在 MySQL 里使用 Sphinx

Sphinx (<http://www.sphinxsearch.com>) 是一个免费的开源的全文搜索引擎，它设计时着眼于能跟数据库完美集成，所以它的功能特性跟 DBMS 类似：快速、支持分布式搜索以及具备伸缩。它也被设计为能够高效地利用内存和磁盘 I/O，这些资源在大规模操作时往往会成为制约性因素。

Sphinx 在 MySQL 上工作得很好，它可以用于加速各种查询，包括全文搜索。你也可以在其他的应用里，将它用作快速分组和排序操作。另外，还有一个插件式的存储引擎可以帮助程序员或管理员通过 MySQL 直接访问 Sphinx。Sphinx 对于某一类查询特别有用，这类查询在 MySQL 通用架构下按照真实世界的设置使用大型数据集时优化得不是很好。简而言之，Sphinx 能够增强 MySQL 的功能和性能表现。

Sphinx 索引的源数据通常就是 MySQL SELECT 查询的结果，但是，你也可以用不同类型的无限的数据来源来建立索引，每一个 Sphinx 示例能搜索到无限的索引。举例来说，你可以从一个位于远程服务器上的 MySQL 实例上拉几份文档放入索引里，再从一个位于另一台远程服务器上的 PostgreSQL 实例上拉几份文档过来，还有几份文档是一个本地的脚本通过 XML 管道机制输出的。

在这个附录里，我们将例举一些能让 Sphinx 体现出性能增强的使用案例，然后讲述一下安装和配置 Sphinx 时所需的主要步骤，接着会从细节上说明它的功能特点，最后讨论几个现实中应用的例子。

C.1 概述：一个典型的 Sphinx 搜索

我们用一个简单但是完整的 Sphinx 应用例子作为进一步讨论的开始。虽然 Sphinx 的 API 可用于各种编程语言，但是，在这里我们使用的是 PHP，因为它比较普及。

假设我们要实现的是一个比较购物引擎里的全文搜索，其具体需求是这样的：

- 在 MySQL 的一个产品表里维护一个可搜索的全文索引。
- 允许使用产品的名称和描述进行全文搜索。
- 如有需要，能够用指定的分类缩小搜索范围。
- 不仅可以按关联度对搜索结果进行排序，也可以用物品的价格或提交日期来排序。

我们先在 Sphinx 配置文件里设置好数据源和索引：

```
source products
{
    type          = mysql
```

```

sql_host      = localhost
sql_user      = shopping
sql_pass      = mysecretpassword
sql_db        = shopping
sql_query     = SELECT id, title, description, \
               cat_id, price, UNIX_TIMESTAMP(added_date) AS added_ts \
               FROM products
sql_attr_uint  = cat_id
sql_attr_float = price
sql_attr_timestamp = added_ts
}

index products
{
    source      = products
    path        = /usr/local/sphinx/var/data/products
    docinfo     = extern
}

```

这个例子假定了 MySQL 上已经建立了购物数据库，其中有一张产品表，表里有供我们执行 SELECT 生成 Sphinx 索引所需的列。Sphinx 索引也是命名后的产品信息。在创建了新的数据库和索引后，我们就运行 imdexer 程序来建立初始化的全文索引文件，然后（重新）启动 searchd 守护进程以同步这些更新。

```

$ cd /usr/local/sphinx/bin
$ ./indexer products
$ ./searchd --stop
$ ./searchd

```

现在索引可用于查询了。我们用 Sphinx 自带的 test.php 示例脚本进行测试：

```

$ php -q test.php -i products ipod

Query 'ipod' retrieved 3 of 3 matches in 0.010 sec.
Query stats:
  'ipod' found 3 times in 3 documents
Matches:
1. doc_id=123, weight=100, cat_id=100, price=159.99, added_ts=2008-01-03 22:38:26
2. doc_id=124, weight=100, cat_id=100, price=199.99, added_ts=2008-01-03 22:38:26
3. doc_id=125, weight=100, cat_id=100, price=249.99, added_ts=2008-01-03 22:38:26

```

625

最后一步是把搜索功能添加到我们的 Web 应用里。我们需要基于用户的输入，设置排序和过滤的选项，好让输出的格式漂亮一些。同时，因为 Sphinx 返回给客户端的只有文档的 ID 和配置属性——它不会存储任何原始文本数据——所以，我们还要亲自从 MySQL 里读出相对应的行数据：

```

1  <?php
2  include ( "sphinxapi.php" );
3  // ... other includes, MySQL connection code,
4  // displaying page header and search form, etc. all go here
5
6  // set query options based on end-user input
7  $cl = new SphinxClient ( );
8  $sortby = $_REQUEST["sortby"];
9  if ( !in_array ( $sortby, array ( "price", "added_ts" ) ) )
10     $sortby = "price";
11  if ( $_REQUEST["sortorder"]=="asc" )
12     $cl->SetSortMode ( SPH_SORT_ATTR_ASC, $sortby );
13  else
14     $cl->SetSortMode ( SPH_SORT_ATTR_DESC, $sortby );
15  $offset = ( $_REQUEST["page"]-1)*$rows_per_page;
16  $cl->SetLimits ( $offset, $rows_per_page );

```

```

17
18 // issue the query, get the results
19 $res = $c1->Query ( $_REQUEST["query"], "products" );
20
21 // handle search errors
22 if ( !$res )
23 {
24     print "<b>Search error:</b>" . $c1->GetLastError ( );
25     die;
26 }
27
28 // fetch additional columns from MySQL
29 $ids = join ( ",", array_keys ( $res["matches"] ) );
30 $r = mysql_query ( "SELECT id, title FROM products WHERE id IN ($ids)" )
31     or die ( "MySQL error: " . mysql_error( ) );
32 while ( $row = mysql_fetch_assoc($r) )
33 {
34     $id = $row["id"];
35     $result["matches"][$id]["sql"] = $row;
36 }
37
38 // display the results in the order returned from Sphinx
39 $n = 1 + $offset;
40 foreach ( $result["matches"] as $id=>$match )
41 {
42     printf ( "%d. <a href=details.php?id=%d>%s</a>, USD %.2f<br>\n",
43             $n++, $id, $match["sql"]["title"], $match["attrs"]["price"] );
44 }
45
46 ?>

```

尽管这段代码看上去相当简单，但是，有一些事情还是值得强调一下：

- SetLimits() 的调用是告诉 Sphinx 只获取客户端要在页面上显示的行号，这就很方便地利用了一下 Sphinx 的限定范围（不像 MySQL 内建的搜索功能）。返回结果的数目可以通过 \$result['total_found'] 来获得。
- 因为 Sphinx 只是索引了 title 列，并没有存储它，所以，它必须从 MySQL 里读取数据。
- 我们使用一条单独的合成查询获取数据，把所有文档 id 都放在 WHERE 字句的 id IN (...) 里，而不是一个文档 id 就运行一次查询（这会很没效率）。
- 我们将从 MySQL 里获取到的行注入到全文搜索的结果集里，保持了原始的排列顺序。在下文里我们会对此作一些解释。
- 我们使用来自 Sphinx 和 MySQL 的数据来显示每一行。

那些由 PHP 写的行注入代码需要再作一些解释。我们不能简单地对 MySQL 查询的结果集作遍历，因为行的次序跟 WHERE id IN (...) 子句里指定的次序不一样（多数情况下都是不一样的）。但是，PHP 会对结果进行散列（使用关联数组），保持了匹配结果插入时的排列顺序，然后 Sphinx 就可以通过 \$result["matches"] 返回排序正确的行了。因此，为了保证来自 Sphinx 的匹配结果能保持正确的次序（不是 MySQL 生成的那种半随机的次序），我们需要把 MySQL 的查询结果一个接一个地注入到 PHP 用来存储 Sphinx 匹配结果集的散列表里。

在计算匹配数目和应用 LIMIT 子句时，MySQL 和 Sphinx 在实现方式及性能上存在着比较大的区别。首先，LIMIT 在 Sphinx 里开销是比较低的。设想有一个 LIMIT 500, 10 子句，MySQL 会半随机地读出 510 行数据（这是比

较慢的)，然后丢弃掉其中的 500 行；而 Sphinx 只会返回一组 ID，你用这些 ID 从 MySQL 上读取到实际所需的数据行。其次，Sphinx 总是能返回它在结果集里找到的实际匹配数目，不管 LIMIT 子句是怎么样的，MySQL 无法做到这么有效（具体可以查看第 194 页的“优化 SQL_CALC_FOUND_ROWS”）。

C.2 为什么要使用 Sphinx

Why Use Sphinx?

Sphinx 可以在多个方面完善基于 MySQL 的应用程序，能补充 MySQL 的性能不足，还提供了 MySQL 所没有的功能。典型的使用场景包括：

- 快速、高效、可伸缩的全文搜索；
- 能在使用低选择性索引或无索引的列时，优化 WHERE 条件语句；
- 优化 ORDER BY ... LIMIT N 查询以及 GROUP BY 查询；
- 并行地产生结果集；
- 向上扩展和向外扩展；
- 聚合分片数据。

我们在下面这些小节里，对这些场景逐一进行探讨。这个列表也不是完整的，Sphinx 的用户还会发现新的应用方法。例如，Sphinx 最重要用途之一——快速扫描和过滤记录——就是由一位用户创造出来的，这可不是 Sphinx 最初的设计目标。

C.2.1 高效、可伸缩的全文搜索

Efficient and Scalable Full-Text Searching

MySQL 的全文搜索功能（注 1）对于小数据集还有点用处，随着数据变大，它的效能就急剧下降。如果索引文本有百万条甚至超过 1GB，查询时间会在 1 秒钟到 10 分钟之间，这对于高性能 Web 应用来说是无法接受的。虽然通过把数据分布到多个地方，可能可以提高 MySQL 全文搜索的性能，但是，这也要求你在应用里并行地执行查询，之后再合并搜索结果。

Sphinx 的运作速度要明显快于 MySQL 内建的全文索引。比如说，它查询 1GB 的文本数据，所需时间在 10 到 100 毫秒之间——在每 CPU 10~100GB 之间呈线性增长。Sphinx 还有以下这些优点：

- 它能对 InnoDB 及其他存储引擎里的数据进行索引，而不仅仅是 MyISAM。
- 它能对多表综合后的数据创建索引，不限于一个单独表上的字段。
- 它能将来自多个索引的搜索结果进行动态整合。
- 除了能对文本列索引外，它的索引还可以包含无限数量的数字属性——跟“额外字段”一样。Sphinx 的属性可以是整型、浮点型和 Unix 时间戳。
- 它能根据属性上的附加条件对全文搜索进行优化。

注 1：请查阅第 244 页的“全文搜索”。

- 基于短句的排位算法能帮助它返回更加切题的结果。举例来说，当你使用 “I love you, dear” 去搜索一首歌的歌词，那么恰好包含了该短句的歌词将被放在最上面，居于其他只是多次包含 “love” 或 “dear” 的歌词之前。
- 它使得向外扩展非常容易。关于扩展的更多内容，请查看第 9 章，或者本附录第 632 页的 “伸缩”。

C.2.2 高效使用 WHERE 子句

Applying WHERE Clauses Efficiently

有时你要在很大的表上（有几百万条记录）做 SELECT 查询，而 WHERE 条件里使用的几个列缺少索引的选择性（例如指定 WHERE 条件返回了太多的行）或者根本没有索引的支持。此类常见的例子有：在一个社交网站上搜索用户，以及在一个拍卖网站上搜索物品。典型的搜索界面是让用户能在 WHERE 条件加入 10 个或更多的列，而返回结果又是按其他列来排序。本书第 131 页的 “索引实例研究” 就是这样一个例子。

当有着合适的数据结构和查询优化时，只要 WHERE 子句不包含太多的列，MySQL 应付这些查询时的工作效率还是可以接受的。但是，当列的数目增加时，支持所有可能的搜索所需的索引数也会随之呈指数级增长。单是要覆盖到 4 个列的所有组合情况，MySQL 就要达到极限了，它变得非常缓慢，并且要花费很多系统开销去维护索引。这意味着对于许多 WHERE 条件，它实际上不可能拥有所需要的所有索引，你不得不在没有索引的条件下运行查询。

更重要的是，即使你能增加索引，它们也无法受益很多，除非它们具备可选择性。有一个典型的例子是 gender 列，它几乎帮不上忙，因为它只能过滤掉所有行的一半。当索引因缺少选择性而无法用于过滤时，MySQL 会回来进行全表扫描。

Sphinx 运行这类查询的速度比 MySQL 快很多。你可以只将数据中所需要的列做成 Sphinx 索引，随后，Sphinx 会允许两种方式来访问这些数据：用关键字作索引搜索或者全扫描。在这两种方式里，Sphinx 都用到了 filters，它相当于就是一个 WHERE 子句。但是，MySQL 是在内部决定使用索引还是全扫描，而 Sphinx 是让你自己选择要使用哪一种访问方法。

要使用带过滤的全扫描，你可以把一个空字符串用作搜索查询条件；要使用索引搜索，你可以在构建索引时加一些伪关键字进去，然后再搜索那些关键字。举例来说，如果你想搜索 123 分类里的物品，你可以在建索引时把 “123 分类” 关键字添加到文档里，然后针对 “123 分类” 做全文搜索。你或者使用 CONCAT() 函数把关键字加入已存在的一个字段里，或者为了更好的兼容性，为这些伪关键字创建一个特别的全文搜索字段。通常而言，对于值的无选择性覆盖率超过 30% 的，你应该选择过滤；而对于可选择性的值覆盖面不超过 10% 的，你应该使用伪关键字；如果目标值是处于 10%~30% 的灰色区域，这个就难说了。你应该做几次基准测试找出最佳解决方案。

Sphinx 无论是执行索引搜索还是全部扫描都快过 MySQL，有时，Sphinx 的全扫描实际上比 MySQL 的索引读取还要快。

C.2.3 找出结果集里的前几行

Finding the Top Results in Order

Web 应用常常需要用到结果集里的前 N 行。如同我们在第 193 页的 “优化 LIMIT 和 OFFSET” 讨论过的那

样，这种操作在 MySQL 里很难被优化。

最糟糕的情况就是根据 WHERE 条件找到了许多行(假设有 100 万行),而 ORDER BY 里的列却没有索引过。MySQL 使用索引识别出所有匹配的行,然后使用半随机磁盘读,把记录一条接一条地读到排序缓冲区里,接着用一种文件排序将这些结果进行排序,最后丢弃其中的绝大多数。它会临时存储和处理整个结果集,忽略 LIMIT 子句,搅乱了 RAM。如果排序缓冲区放不下整个结果集,还需要用到磁盘,引发更多的磁盘 I/O。

这里还有一个极端的例子,你可能会认为这在真实世界里很少会发生,但是,事实上,它反映出的问题常常会发生。MySQL 在用于排序的索引方面是有限制的——只使用索引的最左边部分,不支持松散索引扫描 (Loose index scan),它只允许一个单独的范围条件——意味着真实世界里的查询不能受益于这些索引。即使能够受益到,那么,使用半随机磁盘 I/O 来获取行也是一个性能杀手。

要对结果集进行分页,常常需要做形如 “SELECT ... LIMIT N, M,” 的查询,这是 MySQL 的另一个性能问题。它们会从磁盘里读取 N+M 行,由此引发了大量的随机 I/O,浪费了内存资源。Sphinx 通过消除两个主要问题,使这类查询的速度有了显著的提高:

内存使用

Sphinx 对 RAM 的使用有严格限制,这个限制也是可以配置的。Sphinx 也支持结果集的偏移量和大小,这跟 MySQL 的 LIMIT N,M 语法是一样的,但是它还有一个 max_matches 选项,它以每个服务器和每个查询为基础,可控制“查询缓冲区”的大小。

I/O

如果属性是存储在 RAM 里的, Sphinx 就不会做任何 I/O 操作。即使属性是存储在磁盘上, Sphinx 也是通过顺序 I/O 来读取它们,这比 MySQL 使用半随机方式从磁盘获取行要快很多。

通过关联度(权重)、属性值和聚合函数值(当使用 GROUP BY 时)的综合,你能对搜索结果进行排序,这个排序语句的语法跟 SQL 的 ORDER BY 子句类似:

```
<?php
$ccl = new SphinxClient ( );
$ccl->SetSortMode ( SPH_SORT_EXTENDED, 'price DESC, @weight ASC' );
// more code and Query( ) call here...
?>
```

在本例中, price 是用户指定的属性,存储在索引里, @weight 是一个特定的属性,在运行时创建,它包含的是每一个搜索结果估算出来的关联度。你也可以使用算术表达式对结果再进行排序,算术表达式里可以包含属性值、常用数学运算符和函数:

```
<?php
$ccl = new SphinxClient ( );
$ccl->SetSortMode ( SPH_SORT_EXPR, '@weight + log(pageviews)*1.5' );
// more code and Query( ) call here...
?>
```

C.2.4 优化 GROUP BY 查询

若没有 GROUP BY 功能,对常用的类 SQL 子句的支持也将会不完整,因此, Sphinx 也支持 GROUP BY。但是,跟 MySQL 通用的实现方法不一样的是, Sphinx 擅于高效地从 GROUP BY 任务里解析出实际所需的子集。当遇

到以下几种情况时，这个子集可以覆盖由大数据集（1~100 百万行）生成的报告：

- 结果只是分组行里的一“小”部分（这里的“小”就是 10 万到 100 万行）。
- 当许多分组从分布于服务器集群各处的数据那里获取过来时，它需要很高的执行速度，同时，近似的 COUNT(*) 结果也能够被接受。

实际上，这没有听起来那么严格。第 1 个场景实际上是覆盖了所有能想象到的基于事件的报表。举例来说，对于一小时生成一份的明细报告，经过 10 年之后，就会返回 90,000 条记录。第 2 个场景用平常的语句来表述，就像是“尽可能快而准确地在一个具有 100 百万行的分块表里找到 20 条最重要的记录”。

有两种类型的查询可以加快常用的查询，但是，你也能把它们用在全文搜索的应用上。许多应用需要显示的不仅是匹配的全文，还要显示一些聚合的结果。举例来说，许多搜索页面会显示出在每一个产品分类里找到了多少个匹配项目，或者显示出一张按时间顺序的匹配文档数量变化图。另外一个常见的需求就是对结果进行分组，并显示出每一个分类里关联度最高的匹配项。

631 Sphinx 的 group-by 支持可以让你结合分组和全文搜索，由此可以省去在你的应用或 MySQL 里做分组的开销。

Sphinx 在排序、分组时都使用固定大小的内存，它的效率比基于 RAM 内数据集的类似 MySQL 查询高一些（10% 到 50%）。在这种情况下，Sphinx 的强大能力绝大多数来自于分散负载，极大地降低了延时。对于无法放入 RAM 的巨大数据集，你可以构建一个特别的基于磁盘的索引用于报告，并使用在线属性（在后文里定义）。在这类索引上面的查询执行起来跟磁盘读数据差不多快——在现在的硬件配置上大概每秒 30~100MB。对于这种情况，性能也能过好 MySQL 很多倍，尽管这结果只是一个近似值。

Sphinx 跟 MySQL 的 GROUP BY 最重要的区别在于 Sphinx 在某种条件下，只会产生近似的结果。这有两个原因：

- 分组使用的是一块固定大小的内存。如果有太多的分组要放入 RAM 里，那么匹配时就会按照某个“不幸”的次序进行，每个分组的记录数可能会比实际值小一点。
- 一个分布式搜索只发送聚合后的结果，而不是在节点间发送各自匹配的结果。如果在不同的节点间有了重复的记录，那么，每个分组里的唯一性记录数会大于实际值，因为能删除重复记录的信息不会在节点间传输。

在实际应用中，快速的近似分组数常常是可以被接受的。如果这不能被接受，那么它通常也可以通过调整守护程序和客户端应用程序得到精确的结果。

你也可以生成与 COUNT(DISTINCT <attribute>) 等价的功能。例如，在一个拍卖网站上，你可以使用它来计算每个分类里卖家的确切数目。

最后，Sphinx 可以让你选取一个标准，然后用这个标准在每一个分组里找到单一的“最合适”的文档。举例来说，当以域分组，并以每个域里的匹配数目排序时，你可以从每一个域里选择相关度最高的文档。这在 MySQL 里不使用一个复杂的查询是不可能做到的。

C.2.5 生成并行结果集

Generating Parallel Result Sets

Sphinx 可以让你基于相同的数据同时生成几个结果，并使用固定大小的内存。作为对比，在传统的 SQL 方法里，

你或者是运行两次查询（还要希望在两次运行之间那些数据还待在缓存里），或者创建一个临时表用于存放每次查询的结果集，这方法显然是更好一点。

举例来说，假设你需要针对每天、每星期、每月定期生成该段时间周期里的报表，要 MySQL 生成这些报表，你不得不使用不同的 GROUP BY 子句运行 3 次查询，还要把源数据处理 3 次。然而，Sphinx 对于这些数据只要处理一次就行了，然后就可以并行地生成全部三份报表。

Sphinx 用一个 multi-query 查询机制来做这个工作，不是一个接一个发出查询。你把几个查询做成一个批处理，然后在一个请求里提交它们：

```
<?php
$client = new SphinxClient ( );
$client->SetSortMode ( SPH_SORT_EXTENDED, "price desc" );
$client->AddQuery ( "ipod" );
$client->SetGroupBy ( "category_id", SPH_GROUPBY_ATTR, "@count desc" );
$client->AddQuery ( "ipod" );
$client->RunQueries ( );
?>
```

Sphinx 会分析这个查询，识别出合成的各部分查询，然后并行地执行这些查询。

举例来说，Sphinx 可能注意到：在这个请求里，排序和分组的模式有些不同，而查询是相同的。这就是上面显示的示例代码里的情形——用 price 排序，用 category_id 分组。Sphinx 会创建几个排序队列来处理这些查询。当运行这些查询时，它会一次性地获取到行，然后把它们提交到所有的队列里。再一个接一个运行查询进行比较，消除几个冗余的全文搜索或全部扫描的操作。

要注意并行结果集的生成，虽然这是常见又重要的优化，但是，这是更一般化的 multi-query 机制的一个特例。它不是唯一可能的优化。其中的经验法则是尽可能地把多个查询放在一个请求中，这通常有助于 Sphinx 开展内部优化操作。哪怕 Sphinx 无法并行处理这些查询，它仍然会节省网络的往返通信。如果将来 Sphinx 加入了更多的优化功能，你的查询就能自动地使用到它们，无须做任何修改。

C.2.6 伸缩

Sphinx 的伸缩能力无论在水平方向（向外扩展）还是垂直方向（向上扩展）上都非常好。

Sphinx 完全可以在各机器之间分布。我们上面提到过的用户案例都能受益于工作量跨 CPU 分配。

Sphinx 的搜索守护进程（Searchd）支持特别的分布式索引（Distributed Indexes），后者知道哪些本地和远程的索引需要查询和聚合。这意味着向外扩展就是一次轻微的配置更改。你只需在各个节点间将数据分块，然后配置主节点，使它能向其他节点并行地发出查询。这就是所有要做的事情。

你也能向上扩展，在单独的机器上增加更多 CPU 或内核，从而提高响应速度。你可以在一台单独的机器上运行好几个 searchd 实例，然后通过分布式索引，从另外的机器过来查询它们。还有一种可选择的方法是你可以把一个单独的实例配置为能与它自己通信，这样并行的“远程”查询其实就发生在同一台机器上，但是使用的是不同的 CPU 或内核。

换句话说，使用 Sphinx 的单独查询也能使用到不只一个 CPU（多重并发查询会自动使用多个 CPU）。这跟 MySQL 有显著的区别，MySQL 的一个查询只能使用到一个 CPU，无论有多少个 CPU 可供使用。另外，Sphinx 在并发

执行查询时不需要任何同步，这就避免了使用互斥（一种同步机制）。而互斥正是 MySQL 在多 CPU 环境下才会出现的声名狼藉的性能瓶颈之一。

向上扩展的另一个重要方面是扩展磁盘 I/O。不同的索引（包括部分更大型的分布式索引）能够轻松地放在不同的物理磁盘或 RAID 分卷上以提高响应时间和吞吐量。这个方法的一部分好处跟 MySQL 5.1 的分片表一样，后者能将数据进行分片存储到不同的位置上。然而，分布式索引还有一些超过分片表的优点。Sphinx 使用分布式索引不仅可以分摊负载，还能并行地处理一个查询的各个部份。作为对比，MySQL 的分片表能通过对分片的剪枝来优化一些查询，但查询的处理不是并行的。另外一方面，即使 Sphinx 和 MySQL 分片都能提高查询的吞吐量，如果你的查询是 I/O 密集的，你也可以使用 Sphinx 让所有查询的响应速度得到线性的提高，而 MySQL 分片只能对那些可采用剪枝方式优化的查询有提高的功效。

分布式搜索的工作流程非常直观：

1. 向所有远程服务器发出远程查询。
2. 执行连续的本地索引搜索。
3. 从每个远程服务器上读取到局部搜索结果。
4. 将所有局部搜索结果合并成最终结果集，并将它返回给客户端。

如果你的硬件资源允许的话，你也能在同一台机器上并行地使用几个索引进行搜索。如果有多个物理磁盘驱动器和多个 CPU 内核，那么并发查询就能互不妨碍地执行。你可以假装有一些索引是远程的，然后配置 `searchd`，使它能联系到它自己，于是，就能在同一台机器上发起一个并行查询了：

```
index distributed_sample
{
    type = distributed
    local = chunk1 # resides on HDD1
    agent = localhost:3312:chunk2 # resides on HDD2, searchd contacts itself
}
```

634 从一个客户端的视角来看，分布式索引跟本地索引完全没什么两样。这就允许你使用节点来代理其他的节点集，从而创建出一棵分布式索引的“树”。举例来说，第一级节点可以代理针对第二级上大量节点的查询请求，这样，就可以依次先搜索本地索引，而后把查询传递给其他节点，直到任意深度。

C.2.7 聚合共享数据

Aggregating Sharded Data

构建一个可伸缩的系统常常要涉及数据在不同物理 MySQL 服务器间的分块（分片）。这个问题我们在第 417 页的“数据分块”里已经深入讨论过了。

当数据以合适的粒度分块后，即使一次简单的使用选择性 `WHERE`（这应该非常快速）获取数据行的查询也意味着要关联到许多服务器，还要检查错误，最后在应用里将搜索结果合并在一起。Sphinx 减轻了这种痛苦，因为所有必要的功能都在搜索守护进程里实现了。

考虑这样一个例子：有一个 1TB 大小的表，其中有 10 亿篇博客文章，通过用户 ID 分块到 10 个物理 MySQL 服务器上，这样给定用户的文章总是在同一台服务器上。只要查询是限制在一个用户上，一切都很好：我们根据用户 ID 先选定服务器，然后按常规操作。

现在，假定我们要实现一个归档分页功能，它要显示该用户所有朋友的文章。那么，我们怎么显示按发布日期排序的第 50 页上 981 到 1000 条记录呢？这些朋友的数据很可能是在不同的服务器上。如果仅有 10 个朋友，那就有约 90% 的可能会用到 8 台服务器，如果是 20 个朋友，这可能性就提高到了 99%。因此，对于大多数查询而言，我们需要关联到所有服务器。更糟糕的是，我们还要从每台服务器上取得 1 000 篇文章，然后，在应用程序里综合起来进行排序。按照第 10 章及其他地方所提供的建议，我们将数据裁减到只需要文章 ID 和时间戳。但是，仍然有 10 000 条记录要在应用程序里排序。许多现代脚本语言都会在排序这一步骤上消耗掉大量的 CPU 时间。另外，我们或者要按顺序从每个服务器上获取结果记录（这个会很慢），或者编写一些代码搞一个并行查询线程（这个很难实现和维护）。

在这样的情形下，采用 Sphinx 将比重新发明轮子显得更有意义。在本例中所有要做的事情就是建立几个 Sphinx 实例，从每个表里映射出经常访问的文章属性——在这里就是文章 ID、用户 ID 和时间戳——然后在主 Sphinx 实例上查询全部 981 到 1 000 的记录，并按照发布日期排序。全部算起来大概是 3 行代码。这是更灵活的扩展方法。

C.3 架构概要

图 C-1 展示了 Sphinx 的架构。

Sphinx 是一整套独立的程序。其中，有两个主要程序：

Indexer

这个程序是用来从各种特定的资源上（例如 MySQL 的查询结果）获取到文档，并据此创建全文索引。这是一个后台批处理任务，网站会定时运行它。

searchd

这个程序是一个守护进程，用于查询 indexer 构建的索引。它为应用程序提供了运行时支持。

Sphinx 的发布包里还包含了一套原生的 searchd 客户端 API。这套 API 由多种编程语言写成，在本书写作的时候，这些语言包括 PHP、Python、Perl、Ruby 和 Java。发布包里还有一个 SphinxSE，它作为一个插件式存储引擎的客户端实现，可在 MySQL 5.0 及更新的版本上使用。这套 API 和 SphinxSE 都允许客户端应用连接到 searchd，然后把查询语句传递过去，最后取回搜索结果。

每一个 Sphinx 全文索引都可以跟数据库里的一个表相比较，不同的是表里放置的一行行数据，而 Sphinx 索引包含的是文档。（Sphinx 也有一个独立的数据结构叫做多值属性（Multivalued Attribute），这在下文里会讲到。）每一个文档都有一个唯一的 32 位或 64 位的整数标识符，它是取自数据表里的索引字段（例如，它可以是一个主键列）。另外，每一个文档拥有一个或多个全文字段（每一个都对应于数据库里的一个文本字段）和数值属性。就像一个数据表一样，Sphinx 索引在所有文档里都有着一样的字段和属性。表 C-1 显示了数据表和 Sphinx 索引的相似之处。

表 C-1：数据库结构和相应的 Sphinx 结构

| Database structure | Sphinx structure |
|--|---|
| <pre>CREATE TABLE documents (id` int(11) NOT NULL auto_increment, title` varchar(255), content` text, group_id` int(11), added` datetime, PRIMARY KEY (id));</pre> | <pre>index documents document ID title field, full-text indexed content field, full-text indexed group_id attribute, sql_attr_uint added` attribute, sql_attr_timestamp</pre> |

Sphinx 不会存储来自数据库的文本字段，它只是用它们来建立一个搜索索引。

C.3.1 安装概要

Installation Overview

Sphinx 的安装非常简单，通常包括了以下几个步骤：

1. 从源代码开始构建程序：

```
$ configure && make && make install
```
2. 创建一个配置文件，在文件里定义好数据源和全文索引。
3. 初始化索引。
4. 启动 searchd。

之后，搜索功能就立即可以用于客户端程序了：

```
<?php  
include ( 'sphinxapi.php' );  
$cl = new SphinxClient ( );  
$res = $cl->Query ( 'test query', 'myindex' );  
// use $res search result here  
?>
```

唯一还没做的事情就是定时运行 `indexer`，更新全文索引数据。在重建索引的时候，`searchd` 当前正在使用的索引还是全部可以使用的：`indexer` 会在索引正在使用时删除它们，并创建一个“影子”索引来代替。在索引创建完之后，它会通知 `searchd` 使用这个新的副本。

全文索引是以一种特别的“单一”结构保存在文件系统里（保存路径在配置文件里指定）的，它不适合做增量更新。通常的更新索引的方法就是全部重建。这问题没有看起来那么大，原因有以下几点：

- 创建索引的速度很快。在现在这种硬件设备上，Sphinx 索引普通文本（不带 HTML 代码）的速度是每秒 4~8MB。
- 如同要在下一小节里讲到的那样，你可以把数据分割到几个索引里。重建索引只在需要更新的那部分数据上运行 `indexer`。
- 无须对索引做“碎片整理”——它们本来就是用优化的 I/O 构建的，这能提高搜索速度。
- 数值属性可以直接更新，无须重建全部索引。

在未来的版本里，还会提供一个额外的索引后端，它将支持实时的索引更新。

C.3.2 典型的分片使用方法

Typical Partition Use

让我们把分片讨论得更具体一些。最简单的分片式样就是 `main + delta`，在这里会创建两个索引，用它们对一个文档集进行索引。`main` 索引的是整个文档集，而 `delta` 索引的是那些自上次主索引建立以来发生过变动的文档。

这个设置能完美地匹配有许多数据更新的模式。论坛、博客、电子邮件和新闻存档，以及垂直搜索引擎都是很好的例子。在分类库里的绝大多数数据在第一次放入后就再也没有更改过，通常情况下，只有一小部分的文档会经过更改或者加入。这意味着 `delta` 索引会比较小，可以根据实际需要经常性地重建（例如每隔 1~15 分钟执行一次）。这相当于是对新增的行进行索引。

你无须为了更改文档的关联属性而重建整个索引——你可以使用 `searchd` 在线进行修改。你要标记某一行已被删除，只要在 `main` 索引里设置一个“删除”属性就行了。因此，你可以通过标记 `main` 索引里的文档属性来进行更新，然后再重建 `delta` 索引。要搜索所有不带“已删除”标记的文档就能返回正确的结果。

要知道索引数据可以是任何 `SELECT` 语句的结果，不必只源自一个单独的 `SQL` 表。对于 `SELECT` 语句也没有任何限制。这意味着你可以在结果集被索引之前先在数据库里预处理一下：动态创建一些额外的字段，或者从索引里去除一些字段，或者生成一些值。

C.4 特别的功能特性

Special Features

包括“只是”索引和搜索数据库内容在内，Sphinx 还提供了一些特别的功能特性。以下就是这些功能里最重要的一部分：

- 搜索和排位算法能记录词语的位置，文档内容里跟查询短语相近的词也会被计算在内。
- 你可以把数值属性绑定到文档上，包括多值属性。
- 你可以用属性值进行排序、过滤和分组。
- 你可以通过凸显搜索查询关键字来创建文档片段。
- 你可以在多个机器间进行分布式搜索。
- 你可以优化查询，在同一份数据上生成多个结果集。
- 使用 SphinxSE，你可以在 MySQL 里访问搜索结果。
- 你可以调节 Sphinx 施加在服务器上的负载。

其中的一些我们已经在前文里提到过了，这一节将讲解其余部分。

Sphinx 能记住每一个文档内词语的位置，就像其他开源全文搜索系统那样。但是，跟它们不同的是，它是使用位置对匹配度进行排序，从而，返回更加相关的结果。

有许多因素会影响到文档的最终排位。要计算排位，别的许多系统只使用了关键字的频度：每一个关键字的出现次数。被所有全文搜索系统使用的经典的 BM25 权重函数（注 2）就是把更多的权重分给那些词语，它们或者在特定的文档里常常出现，或者在整个文档集里很少出现。BM52 返回的结果通常就是最终的排位值。

相反，Sphinx 也计算了查询短语的近似度，它仅仅是文档内查询子短语的最多字数的数值，以词语为计数单位。举例说来，用短语“John Doe Jr”在带有文本“John Black, John White Jr, and Jane Dunne”的文档里搜索时，会产生一个短语近似度 1，因为按查询序列，没有两个词语同时出现在文档里。如果文档的文本是“Mr. John Doe Jr and friends”，那就是近似度 3，因为这三个查询词语依次出现在文档里。而文本“John Gray, Jane Doe Jr”会生成近似度 2，这是因为“Doe Jr”是查询子短语。

在默认情况下，Sphinx 首先是用短语近似度排序的，而经典 BM52 的权重在其次。这意味着逐字的查询引用保证可以在最上面，如果一个单独的词语刚好在它们下面，那引用就会被关闭。

短语近似度是何时又是如何影响结果的呢？设想要在 1 000 000 页的文本里搜索短语“To be or not to be”。Sphinx 会将页面用逐字引用的方法放在搜索结果的最前面，而其他基于 BM52 的系统会首先返回那些包含了最多的“to”、“be”、“or”和“not”的页面——那些包含了确切引用的页面只因里面的“to”不够多，就被埋藏在搜索结果的深处了。

当今许多主流的 Web 搜索引擎用关键字的位置对结果进行排位也是一样的原理。在 Google 上搜索一个短语，它就会把最完美的或接近完美的匹配短语放在结果的最上面，后面跟随的是“词袋子”文档。

然而，分析关键词的位置会需要额外的 CPU 时间，有时你可能会出于性能考虑而跳过这一步。在有些情况下，短语排位也会产生不受欢迎的、出乎意料的结果。举例来说，在一块 tag 云里搜索 tag 时没有关键字位置会更好一些：要查询的 tag 在文档里是否相邻也没什么区别。

为了顾及兼容性，Sphinx 提供了排位模式的选择。除了默认的近似度加 BM25 之外，你还能选用多种其他类型的方法，包括只有 BM25 权值的、全面禁用权值的（如果你不是使用排位做排序的话，它能提供很好的优化操作）等等。

每一个文档都可以包含无限数目的数值属性。属性是用户指定的，能够根据特定任务的需要，容纳任何额外的信息。这样的例子如一篇博客文章的作者 ID、明细表里一个项目的价格、一个分类 ID 等等。

属性使全文搜索能利用额外的过滤和排序，提高搜索的效率，并能对搜索结果进行分组。在理论上，它们可以被存储在 MySQL 里，在搜索执行时再取出来。但是，在实际应用里，如果全文搜索要在 MySQL 里定位几百或几千行数据（这也不算多）并读取出来，那将是不可接受地慢。

注 2：更多内容请访问 http://en.wikipedia.org/wiki/Okapi_BM25。

Sphinx 支持两种方式存储属性：内联到文档列表里或者放在外部一个单独的文件里。内联要求每当有文档 ID 存入的时候，所有属性值要在各索引里存储多次。这会增加索引的大小，还会提升 I/O 数量，但是，它也减少了 RAM 的使用。在外部对属性进行排序需要在 searchd 启动时，把它们预加载到 RAM 里。

通常，属性都能被放入 RAM 中，因此，常见的做法就是把它存储在外部。这可以使过滤、排序和分组更加快速，因为访问这些数据就相当于是一次快速的内存内查找。同时，只有存放于外部的属性才能在运行时被更新。内联存储应该只在没有足够的空闲 RAM 来存储属性数据时才可以被使用。

Sphinx 也支持多值属性（Multivalued Attributes, MVA）。MVA 的内容由一个任意长的整数值列表组成，每个整数值对应一个文档。那些善于利用 MVA 的例子有 tag ID 列表、产品的分类和访问控制列表。

C.4.3 过滤

Filtering

在全文搜索引擎里拥有对属性值的访问权可以让 Sphinx 在搜索时尽可能早地过滤、剔除候选匹配项。从技术上来说，过滤检查是在校验完文档是否包含了所有需要的关键字之后，又在某个计算量很大的计算过程（例如排名）之前。因为有了这些优化，使用 Sphinx 把过滤和排序整合到全文搜索里要比 Sphinx 搜索、MySQL 过滤这一过程快 10 到 100 倍。

Sphinx 支持两种类型的过滤，这与 SQL 里简单的 WHERE 条件很相似：

- 一个属性值匹配一个特定范围内的值（跟 BETWEEN 子句相似，或者像是数值比较）。
- 一个属性值匹配一个特定的值集合（跟 IN() 列表相似）。

640

如果过滤器有固定数目的值（用“集合”过滤器代替了“范围”过滤器），如果这些值是可选择性的，那么，用“伪关键字”替换掉整型值，并用内容的全文搜索代替属性的搜索是很有意义的。这同样适用于普通的数值属性和多值属性。在下文里我们会看到一些关于如何去做的例子。

Sphinx 能够使用过滤器来优化全文扫描。它能记住在一小段连续的行块（默认是 128 行）中最小和最大属性值，这样，Sphinx 就能根据过滤条件很快地丢弃掉不符合条件的数据块。存储在数据块里的行是按照文档 ID 降序排列的，因此，这个优化工作最适合那些跟 ID 关联紧密的列。例如，如果你有一个行插入时间戳，它会随着 ID 一起增长，那么，在这个时间戳上做带有过滤的全扫描会非常的快。

C.4.4 SphinxSE 插件式存储引擎

MySQL SphinxSE Plugin Storage Engine

接收到来自 Sphinx 的全文搜索结果之后，几乎总会有一些额外的涉及到 MySQL 的工作要做——从最低限度来讲，Sphinx 索引里没有存储的文本列的值必须从 MySQL 那里取得。由此带来的结果是，你会经常需要把 Sphinx 的搜索结果和其他 MySQL 表联接起来。

虽然你也可以通过把搜索结果里的文档 ID 写入一条查询语句发送给 MySQL 执行，但是，这种方法会导致产生既不清晰也不快速的代码。遇到高流量的局面时，你就应该考虑使用 SphinxSE——一个插件式的存储引擎。你可以把它编译到 MySQL 5.0 或更新的版本里，或者作为一个插件加载到 MySQL 5.1 或更新的版本里。

SphinxSE 可以让程序员从 MySQL 里面查询 searchd，访问搜索结果。这个用法非常简单，只要用带有 ENGINE=SPHINX 子句的方法创建一个特殊的表就行了(还有一个可选的 CONNECTION 子句能够在 Sphinx 服务器不在默认位置上时重新定位服务器)，然后就可以在表上运行查询了：

```
mysql> CREATE TABLE search_table (
->   id          INTEGER NOT NULL,
->   weight      INTEGER NOT NULL,
->   query       VARCHAR(3072) NOT NULL,
->   group_id    INTEGER,
->   INDEX(query)
-> ) ENGINE=SPHINX CONNECTION="sphinx://localhost:3312/test";
Query OK, 0 rows affected (0.12 sec)

mysql> SELECT * FROM search_table WHERE query='test;mode=all' \G
***** 1. row *****
      id: 123
     weight: 1
    query: test;mode=all
   group_id: 45
1 row in set (0.00 sec)
```

每一个 SELECT 会把 Sphinx 查询语句放在 WHERE 子句的 query 列里传递出去，Sphinx searchd 服务器就会返回查询结果，然后 SphinxSE 存储引擎会把它们翻译成 MySQL 的结果并返回给 SELECT 语句。

其中的查询可能会包含 JOIN，把别的存储引擎上的其他表联接进来。

SphinxSE 支持许多原本在 API 里才可用的搜索选项。设定选项的方法就是把额外的语句加入到查询字符串里，就像过滤和范围限制那样：

```
mysql> SELECT * FROM search_table WHERE query='test;mode=all;
-> filter=group_id,5,7,11;maxmatches=3000';
```

通过 API 返回的每一个查询和每一个词语的统计信息也可以用 SHOW STATUS 访问到：

```
mysql> SHOW ENGINE SPHINX STATUS \G
***** 1. row *****
      Type: SPHINX
      Name: stats
Status: total: 3, total found: 3, time: 8, words: 1
***** 2. row *****
      Type: SPHINX
      Name: words
Status: test:3:5
2 rows in set (0.00 sec)
```

甚至当你在使用 SphinxSE 的时候，经验法则仍然允许 searchd 执行排序、过滤和分组——也就是说，把所有需要的子句加入到查询字符串里，要好过使用 WHERE、ORDER BY 和 GROUP BY。这对于 WHERE 条件来说尤为重要，其原因是 SphinxSE 仅仅是 searchd 的一个客户端，不是一个全能的内嵌搜索库。然而，你还是要把所有东西都传递给 Sphinx 引擎，以获得最好的性能表现。

C.4.5 高级性能控制

Advanced Performance Control

包括索引和搜索操作都会显著地加重搜索服务器或数据库服务器的负担。幸运的是，有很多设置可以让你限制

来自 Sphinx 的负载。

Indexer 的查询会引发非期望的数据库端负载，它们或者是因为使用到的锁彻底减慢了 MySQL 的运转速度，或者是因为出现得太快而与其他并发查询竞争资源。

第 1 个例子就是 MyISAM 的一个声名狼藉的问题，当长时间地读锁定表时，会影响到其他后续的读和写——你只要在服务器上执行一下 `SELECT * FROM big_table` 就知道了：

```
sql_query_range = SELECT MIN(id),MAX(id) FROM documents
sql_range_step  = 1000
sql_query       = SELECT id, title, body FROM documents \
                  WHERE id>=$start AND id<=$end
```

这个结构在对 MyISAM 表索引的时候非常有用，但是，它也应该考虑到使用 InnoDB 表的情况。虽然 InnoDB 在运行一个大数据量 `SELECT *` 的时候不会锁定表，也不会延误其他查询的执行，但是，由于它的 MVCC 架构，它还是会使用到一些重要的机器资源。如果有 1 000 个多重版本的事务，每个事务会涉及到 1 000 行数据，它们的总开销也会小于单独一个要长时间运行的涉及到 100 万行数据的事务。

第 2 种加重负载的可能发生在 indexer 能够比 MySQL 更快地处理数据的时候。在这样的案例里，你应该已经用到了范围查询。`sql_ranged_throttle` 选项会强迫 indexer 在后续的查询步骤之间休眠一段时间（可由用户指定，以毫秒计算），这虽然增加了索引建立的时间，但是减轻了 MySQL 的负担。

如果你有足够兴趣，还可以看一个特别的案例：你可以用反向的方法来调优 Sphinx。具体讲就是为了缩短索引建立时间，就将更多的负载加到 MySQL 上面。当 indexer 和数据库之间的连接是 100MB 时，那些行都压缩得很好（典型的是文本数据），MySQL 的压缩协议能缩短整体的索引时间。随之而来的是另外一个开销增加了：为了压缩和解压缩网络上传输的行数据，MySQL 端和 indexer 端各自都会使用到更多的 CPU 时间。但是，整个索引时间能够缩短 20%~30%，因为这减少了网络数据流量。

集群上的搜索偶尔也会遇到过载问题，因此，Sphinx 提供一些方法以避免 searchd 跑飞了。

首先，`max_children` 选项可以简单地限制一下能够并发运行的查询数量，如果达到极限了，就告诉客户端重试。

其次，还有查询级别的限制。你可以设定查询运行的时候，或者是在找到预定个数的匹配项时就停止，或者是用完指定长度时间之后就停止。这两个条件分别可以通过调用 `SetLimits()` 和 `SetMaxQueryTime()` API 来实现。这是针对每一个查询设置的，所以，你可以确保更重要的查询总是能够彻底完成。

最后，定时运行 indexer 会引起额外 I/O 的突然增加，随之会影响到 searchd 间歇性地速度减慢。为了防止出现这种情况，Sphinx 里有相应的选项来限制 indexer 的磁盘 I/O。`max_iops` 限制了两次 I/O 操作之间的最长延迟时间，以确保每一秒里不会有超过 `max_iops` 时间的磁盘操作会被执行。但是，有时一个单独的操作也会很占时间，比如有一个 100MB 数据量的 `read()` 调用。`max_iosize` 选项可以关照到这种情况，它保证每一次磁盘读或者写的长度都被限制在指定的范围之内。更大的操作会被自动地分解成小型操作，然后，这些小型的操作被 `max_iops` 设置控制了。

C.5 实际应用案例

Practical Implementation Examples



我们描述过的每一项功能特性都可以找到成功部署的案例。以下几个小节里，我们将回顾几个真实世界里的 Sphinx 实际应用，简要地介绍一下那些站点以及一些实现上的细节。

C.5.1 Mininova.org 的全文搜索

Full-Text Searching on Mininova.org

Mininova.org 是一个广受欢迎的 BT 种子搜索引擎，它为如何优化“纯”全文搜索功能提供了一个清晰的案例。Sphinx 用不能处理负载的 MySQL 内建全文索引替换了那几个 MySQL 从服务器。在替换之后，搜索服务器就能轻装上阵了，当前平均负载率大概在 0.3~0.4 之间。

以下是数据库大小和负载量的数据：

- 网站使用的是一个小型数据库，大约有 300 000~500 000 条记录，索引有 300~500 MB。
- 网站的负载相当高：在本书编写的时候，已经达到约每天八百万到一千万的搜索量。

数据主要是用户提供的文件名，经常会有不正确的标点符号，鉴于此，它用前缀索引替代了全词索引。由此生成的索引会比原来方式生成的大上好几倍，但是，它仍然小到能够被快速地重建，数据也仍然能高效地缓存。

最常见的 1000 个查询的结果会被缓存在应用程序端，所有查询里的 20%~30% 受益于这个缓存。由于查询分布里也有“长尾理论”的存在，更大的缓存也起不了更大的作用。

出于高可用性的考虑，网站使用了两台服务器用作完全的全文索引复制，每过几分钟就将索引重建一次。这个索引过程花费的时间不超过 1 分钟，因此，无须实现更复杂的系统配置。

以下就是从这个案例里学到的经验：

- 在应用里缓存搜索结果会有很大帮助。
- 哪怕是很忙的应用，可能也不需要巨大的缓存。能有 1000 到 10000 条结果记录在缓存里就够用了。
- 对于 1GB 左右的数据库来说，简单的定时重建索引能够很好地替代更复杂的系统配置，甚至是访问量很大的网站也如此。

C.5.2 BoardReader.com 的全文搜索

Full-Text Searching on BoardReader.com

Mininova 是一个负载特别高的项目案例——它的数据并不多，但是对于这些数据有大量的查询。BoardReader (<http://www.boardreader.com>) 起初的样子恰恰相反：它是一个论坛搜索引擎，要在大量的数据集上作一些数量不多的查询。后来，Sphinx 替换了原有的一个商业化全文搜索引擎，那个引擎在 1GB 数据上执行一次查询需要 10 秒钟，而 Sphinx 无论在数据量大小上，还是查询吞吐量上都给予了 BoardReader 极大的扩展空间。

这里有一些说明性信息：

- 在数据库里有超过 10 亿个文档和 1.5TB 文本数据。
- 每天大约有 50 万浏览量，70 万到 100 万次搜索。

在本书正在编写之时，这个搜索集群包括了 6 台服务器，每一台服务器拥有 4 个逻辑 CPU（两个双核 Xeon），16GB 的 RAM，以及 0.5TB 的磁盘空间。数据库放在另外一个单独的集群里，搜索集群只用来建立索引和执行搜索。

这 6 台服务器的每一台都运行着 4 个 `searchd` 实例，因此，所有 4 个 CPU 内核都使用到了，这 4 个实例中的一个用来从其他 3 个那里聚合搜索结果。整个搜索集群里 `searchd` 实例总共是 24 个，数据均匀分布在它们之上。

来自这 6 个“第一层”`searchd` 节点的搜索结果会依次由另一个运行在前端 Web 服务器上的 `searchd` 聚合。这个 `searchd` 实例传递的是纯粹的分布式索引，指向那 6 个搜索集群服务器，没有任何本地数据。

为什么每个节点上要有 4 个 `searchd` 实例？为什么不像我们前面建议过的那样，每台服务器一个 `searchd` 实例，并把它配置为承载 4 个索引块，让它作为一台远程服务器，自己跟自己联系，以充分利用多 CPU 的好处？用 4 个实例代替一个有它特有的好处。首先，它缩短启动时间。每次启动都有好几 GB 的分布式数据需要加载到 RAM 里，同时启动几个守护进程能让我们并行完成这个过程。其次，它提供了可用性。当一个 `searchd` 故障或更新时，那也只是全部索引的 1/24 不能访问，而不是 1/6。

在搜索集群的 24 个实例里，我们使用基于时间的分片技术进一步减少负载量。许多查询只须运行在最近收录的数据之上，因此，数据可以被分成 3 个相互分离的索引集：最近一个星期的数据、最近 3 个月的数据和全部数据。这些索引分布在每个实例所在服务器的几块物理磁盘上。通过这个方法，每个实例都拥有了自己的 CPU 和物理磁盘驱动器，互不干扰。

本地的 `cron` 任务会定时更新索引，它跨网取得 MySQL 数据，但是，是在本地创建索引文件。

使用几块明确独立的“裸”磁盘被证明快于单独的 RAID 卷。裸磁盘能够控制到哪块物理磁盘上的哪一些文件，而同样的情况在 RAID 里，是控制器决定哪一块物理磁盘上的哪一个数据块。裸磁盘也能保证在不同的索引块上充分使用并行 I/O，但是，RAID 上的并发查询仍然受制于步进式的 I/O。我们在此处选择的是没有冗余的 RAID 0，这是因为我们不关心磁盘故障，在搜索节点上重建索引是很方便的事情。当需要提高可靠性时，我们也使用几个 RAID 1（镜像）卷提供跟裸磁盘一样的吞吐量。

从 BoardReader 那里了解到的另一个有趣的事情是 Sphinx 版本升级是如何执行的。显然，整个集群是不可能停掉的，因此，向后兼容非常重要。幸运的是，Sphinx 提供了这样一项功能——新版本的 `searchd` 一般都能读出旧版本的索引文件，也能跨过网络跟旧的客户端通信。要注意的是第一层上用来聚集搜索结果的节点对于第二层节点而言就是客户端，因为是后者实际执行了搜索。所以，升级的次序是：先升级第二层上的节点，然后是第一层上的节点，最后才是 Web 前端。

在本例中学到的经验是：

- 对于超大型数据库的格言是：分片、分片、分片、并行。
- 在大规模的搜索应用里，要将 `searchd` 组织为树状的多层结构。
- 尽可能地针对全部数据的一小部分建立优化过的索引。
- 明确地将文件映射到磁盘要好过依靠 RAID 控制器的实现。

C.5.3 Sahibinden.com 对 Select 的优化

Optimizing Selects on Sahibinden.com

Sahibinden.com 是土耳其一家领先的拍卖网站，原先存在着大量的性能问题，包括全文搜索的性能。在部署了 Sphinx，并对查询作了一些分析之后，最后发现 Sphinx 带有过滤时执行许多常见的跟应用相关的查询要比 MySQL 快得多——甚至当一部分索引是在 MySQL 的一个卷上时。另外，把 Sphinx 用于非全文搜索时，使用的也是统一的应用代码，易于编写和支持。

MySQL 的表现不佳是因为每个单独卷的选择性不足以明显地缩小搜索空间。事实上，要创建和管理所有需要的索引几乎是不可能的，因为太多的卷需要它们。产品信息表大约有 100 个列，在技术上，Web 应用可能会将它们中的每一个都用于过滤或排序。

在这个“热门”的产品表上的主动插入和更新都是慢如龟行，因为太多的索引需要随之更新。

基于这些原因，要应付产品信息表上的所有 SELECT 查询，而不仅仅是全文搜索查询，Sphinx 就是一个自然的选择。

这里是网站数据库的大小和负载量：

- 数据库里包含了大约 400 000 行记录，500MB 的数据。
- 负载量大约是每天 300 百万个查询。

为了模仿普通的带 WHERE 条件的 SELECT 查询，Sphinx 在建立索引过程中把一些特别的关键字写在全文索引里。这些关键字都形如 `_CATN_`，这里的 N 可以用相应的分类 ID 来代替，这个替代过程发生在 MySQL 查询中运行带 `CONCAT()` 函数的索引之时，因此，不会影响到源数据。

索引需要尽可能频繁地重建，在这里是每分钟重建一次。在多 CPU 环境里的每个 CPU 上，每次重建的时间是 9~15 秒，因此，早先讨论过的 `main + delta` 方案是不需要的。

实践证明，PHP API 在解析带有大量属性的结果集时，会花费相当数量的时间（每个查询大约 7~9 毫秒）。通常，这个开销不会构成问题，因为全文搜索的开销，特别在大数据集上执行时，会高于这个解析开销。为了减少这个因素的影响，索引被分隔成一对对的：“轻量”的有 34 个常用的属性，而“完整”的有全部 99 个属性。

另一种可能的解决办法是使用 SphinxSE 或者实现一个功能，它能够把特定的列读到 Sphinx 里。然而，同时考虑到时间，使用两个索引的方法是目前实现起来最快捷的。以下就是我们从这个案例里学到的经验：

- 有时，Sphinx 上的全扫描的执行效率要好过 MySQL 的索引读取。
- 出于选择性条件，用“伪关键字”代替属性过滤之后，全文搜索引擎就能做更多的工作。
- 脚本语言里的 API 在某些极端条件下会成为性能瓶颈，但是真实世界的应用里一般不会。

C.5.4 BoardReader.com 对 GROUP BY 的优化

Optimizing GROUP BY on BoardReader.com

想要改进 BoardReader 服务，就要计算超链接数，要根据关联数据创建不同的报表。例如，报表之一就是要显示出最近一个星期来链接数排在最前面的 N 个二级域名。同样，还可以计算链接到指定站点（例如 YouTube）

的最前面 N 个二级和三级域名。用来创建这些报表的查询具有下列这些常见特征：

- 它们总是用域来分组的。
- 它们的排序是每个组里的链接数，或者是每个组里的唯一域名的链接数。
- 它们要处理大量的数据（接近几百万行记录），但是，最后生成的带有最佳分组的结果集往往又很小。
- 近似的结果也能被接受。

在原型测试环节里，MySQL 执行这些查询大概花了 300 秒。理论上，通过数据分片技术，把它们分拆到各个服务器执行，再在应用里用人工方式聚合查询结果，还可能将这些查询的时间优化到 10 秒左右。但是，这需要构建一个复杂的架构，而且分片的实现也远不是那么直接明了。

因为我们已经成功地使用过 Sphinx 对搜索负载进行分布式处理，所以，我们决定也使用 Sphinx 来实现一个近似的分布式 GROUP BY。这就要求在建立索引之前预处理这些数据，把所有感兴趣的子串转换为单独的“词语”。以下就是一个示例 URL 在预处理前后的样子：

```
source_url      = http://my.blogger.com/my/best-post.php
processed_url    = my$blogger$com, blogger$com, my$blogger$com$my,
                  my$blogger$com$my$best, my$blogger$com$my$best$post.php
```

美元符号 (\$) 仅仅是对 URL 分隔符作了统一的替换，这样，搜索就能用 URL 任何部分（域或者路径）来控制了。这种预处理能析取出所有“感兴趣”的子串成为单独的关键词，这样搜索起来是最快的。在技术上，我们可以采用短语查询或者前缀索引，但是，那些都会导致更大的索引，速度也更慢。

链接的预处理是在构建索引时进行的，使用了特制的 MySQL UDF。在这个任务里，我们还定制了一个计算唯一性值的功能用来加速 Sphinx。之后，我们就能把查询全部移到搜索集群里，分发它们很简单，但是，减少的查询延迟却是很明显。

这里是数据库的大小和负载量：

- 里面约有 1.5~2 亿行记录，在预处理之后，会生成 50~100GB 的数据。
- 负载是每天大概 6~10 万个 GROUP BY 查询。

这个用于分布式 GROUP BY 的索引也是部署在先前我们提到过的同一个搜索集群上——有着 6 台机器，24 个逻辑 CPU。对于存储了 1.5TB 文本的数据库而言，这只是主要搜索负载的一个小小的零头。

Sphinx 用近似、快速、分布式的计算方式替代了 MySQL 的精确、缓慢、单 CPU 的计算方式。所有会引入近似错误的因素都会存在：输入数据常常有太多的行，以至于没法放入“排序缓冲区”里（我们使用的是一个固定的 10 万行的 RAM）、使用了 COUNT(DISTINCT)、结果集是通过网络聚合的。尽管有这些麻烦在，但是对于结果集里的前 10 到 1000 分组——这常常是报表实际所需的——能够有 99%到 100%的准确率。

这些需要索引的数据跟普通全文搜索用的数据很不一样。它里面有巨大数目的文档和关键词，尽管文档都非常小。文档的编号也是非连续的，因为它使用的是一种特殊的编号规则（综合了来源服务器、来源表和主键），所以，无法用 32 位来存储。巨大数量的搜索“关键词”也会引发常见的 CRC32 冲突（Sphinx 用 CRC32 把关键词映射到内部词语 ID）。出于这些原因，我们只能被迫在系统内到处使用 64 位的标识符。

目前，系统的性能很令人满意。对于大多数复杂的域名而言，完成一次查询的时间通常是 0.1 到 1.0 秒。

以下就是从这个案例里学到的经验：

- 对于 GROUP BY 查询，可以为了速度牺牲掉一些精度。
- 对于大量文本集合或者适当大小的特殊数据集，可能要用到 64 位标识符。

C.5.5 Grouply.com 对共享联接 (JOIN) 查询的优化

Optimizing Sharded JOIN Queries on Grouply.com

Sphinx 对 MVA 的支持是一个相当新的功能特性，但是，用户们已经发现了更聪明的用法。Grouply.com 构建了一个基于 Sphinx 的解决方案，用来搜索数据库里那些打了标签的信息（有好几百万条）。出于扩展性的考虑，这个数据库被拆分到多个物理服务器上，因此，它可能需要位于不同服务器上的数据表。任何大规模的联接是不可能实现的了，因为这会涉及到许多服务器、数据和表。

Grouply.com 使用 Sphinx 的 MVA 属性来存储消息标签。这个标签列表是通过 PHP API 从 Sphinx 集群那里获取到的，以此替代了依次执行多个 SELECT 从几个 MySQL 服务器上读取的过程。为了减少 SQL 查询的执行次数，某些只需要呈现的数据（例如，最近读过这条消息的用户列表）也被存储在一个单独的 MVA 属性里，并通过 Sphinx 访问。

649 这里有两项创新使用了 Sphinx 重建 JOIN，还用 Sphinx 的分布式功能归并了分散在各个数据块上的数据。只使用 MySQL 是不可能做到这些的。高效的归并要求数据能够分拆到尽可能少的物理服务器和表上，但是，这又会伤害到伸缩性和可扩充性。

从此例中学到的经验是：

- Sphinx 能够有效地聚合用于高度分区化的数据。
- MVA 能够用于存储和优化预构建的 JOIN 结果。

C.6 结论

Conclusion

在本附录里，我们只是简要地讨论了 Sphinx 全文搜索系统。为了缩短篇幅，我们特意略过了关于 Sphinx 其他许多功能特性的讨论，例如对 HTML 索引的支持、对 MyISAM 有更好支持的范围搜索、词法和同义词的支持、前缀和中缀索引和 CJK 索引。不过，这个附录应该已经给了你一些关于 Sphinx 如何高效解决真实世界各种问题的印象。它也不限于做全文搜索，还能解决许多对于传统 SQL 而言很困难的问题。

Sphinx 既不是一颗银弹，也不是 MySQL 的一个替代品，但是，在许多应用案例里（对于现代 Web 应用已经变得很普通），它可以被当作 MySQL 很有用的补充。你可以用它来减轻一些工作的负担，或者说，甚至能为你的应用创造新的可能性。

你可以在 <http://www.sphinxsearch.com> 下载到 Sphinx——不要忘记分享你自己的用法！

在任何用锁控制资源共享访问的系统里，如突然出现了锁的竞争，那调试起来就很困难。可能正当你往表里新增一列时，或者正在运行一个查询时，突然发现由于有别的什么东西锁住了你正要使用的表或行，你的查询就阻塞在那里了。这个附录讲述的就是当你在 MySQL 里遇到这种情形时该怎么去做。通常，你想到的是找出查询被阻塞的原因，但是，有时也会想知道到底是谁阻塞了它，这样你就知道该杀掉哪个进程了。这个附录将告诉你如何达到这两个目标。

D.1 服务器级锁的等待

Lock Waits at the Server Level

一个锁等待既可能发生在服务器级上，也可能发生在存储引擎级上（注 1）。（应用级上的锁也是一个问题，但是，我们这里只关注 MySQL 上的。）

MySQL 服务器本身使用着好几种类型的锁。如果在服务器级上有一个查询等待一个锁，你可以通过查看 `SHOW PROCESS LIST` 的输出来找到它的痕迹，例如 InnoDB 实现了它自己的锁，至少在它写的时候会用到。在 MySQL 5.0 及更早的版本里，服务器是不知道这一类锁的，它通常都隐藏起来，不让用户和数据库管理员看到。在未来的版本里，这类处于服务器级上的锁可能会通过具有插件功能的 `INFORMATION_SCHEMA` 表越来越多地显露出来。

以下是 MySQL 服务器用到的几种锁：

表锁

在读或者写的时候，表也能被锁住。在这些锁里有一对锁的变体，例如本地读锁。关于这些变体，你可以阅读 MySQL 使用手册的 `LOCK TABLES` 那一节。除了这些显式锁外，在查询的执行过程中还有隐含锁的存在。

全局锁

只有一个全局的读锁，可以在执行 `FLUSH TABLES WITH READ LOCK` 时获取到。

名称锁

名称锁是当服务器重命名或删除一个表时创建的一种表锁。

注 1：如果你要刷新服务器与存储引擎之间那块分离的内存，请参考第 1 章里的图 1-1。

字符串锁

你可以使用 `GET_LOCK()` 和其相关函数锁定和释放任何一个服务器范围内的字符串。

下面我们将对每一种类型的锁逐一作更具体的描述。

D.1.1 表锁

表锁可以是显式的或隐含的。使用 `LOCK TABLES` 创建的就是显式锁。举例来说，如果你在一个 `mysql` 会话里执行下面这个命令，你就会有在 `sakila.film` 上加上一个显式锁：

```
mysql> LOCK TABLES sakila.film READ;
```

如果在此之后，你在另一个不同的会话里执行下面这个命令，这些查询就会被挂起，无法完成：

```
mysql> LOCK TABLES sakila.film WRITE;
```

这时，你可以查看到在第一个连接里的那些等待线程：

```
mysql> SHOW PROCESSLIST\G
***** 1. row *****
  Id: 7
  User: baron
  Host: localhost
  db: NULL
  Command: Query
  Time: 0
  State: NULL
  Info: SHOW PROCESSLIST
***** 2. row *****
  Id: 11
  User: baron
  Host: localhost
  db: NULL
  Command: Query
  Time: 4
  State: Locked
  Info: LOCK TABLES sakila.film WRITE
2 rows in set (0.01 sec)
```

652 要注意线程 11 的状态是锁定。在 MySQL 服务器的代码里，只有一处地方会让线程进入这个状态：当它试着获取一个表锁的时候，已经有另外的线程占有了这个表锁。因而，如果你看到这个信息，你就能知道线程等待的是 MySQL 服务器里的锁，而不是存储引擎里的。

然而，显式锁并不是唯一能阻塞这个操作的锁。如同我们前文里提到过的那样，服务器会在查询时隐含地锁定表。展示这种效果的最简单方法就是运行一个耗时很长的查询——在查询语句里加入 `SLEEP()` 函数，很轻易地就能做到。

```
mysql> SELECT SLEEP(30) FROM sakila.film LIMIT 1;
```

当查询正在运行时，如果你再次试着去锁定 `sakila.film`，这个操作会因为隐含锁的存在而被挂起，就像有显式锁的时候那样。通过进程列表，你可以看到它的执行效果：

```
mysql> SHOW PROCESSLIST\G
***** 1. row *****
    Id: 7
   User: baron
  Host: localhost
    db: NULL
 Command: Query
    Time: 12
   State: Sending data
   Info: SELECT SLEEP(30) FROM sakila.film LIMIT 1
***** 2. row *****
    Id: 11
   User: baron
  Host: localhost
    db: NULL
 Command: Query
    Time: 9
   State: Locked
   Info: LOCK TABLES sakila.film WRITE
```

在本例中，SELECT 查询里的隐含读锁阻塞了 LOCK TABLES 请求的显式写锁。隐含锁之间也会相互阻塞。

你大概正迷惑于显式锁和隐含锁的区别。从内部原理上来说，它们是同一类型的锁，都是 MySQL 服务器代码在控制它们。从外部行为上来说，你可以使用 LOCK TABLES 和 UNLOCK TABLES 来控制显式锁。

然而，当涉及除 MyISAM 之外的存储引擎时，它们之间就有很重要的区别。当你显式地创建一个锁时，它会完成你想让它做的任何事情，但是，隐含锁是隐藏的，“不可捉摸”的。服务器会根据需要自动创建和释放隐含锁，它告诉存储引擎这些隐含锁的存在，存储引擎就会“转换”出合适的锁。举例来说，InnoDB 就有关于为指定的服务器级表锁创建哪种 InnoDB 表锁的规则。这样就很难了解到 InnoDB 在幕后到底创建了什么样的锁。

在 MySQL 5.0 和 5.1 里，服务器管理服务级表锁的方式是死锁-释放：按照内部定义好的次序，在同一时间里创建和释放锁。在 MySQL 6.0 里，在没有释放你已存在的锁之前，也可能加入更多的锁，因此，在表级锁里也可能出现死锁。然而，在本书写作时，这项功能还没有完成，最终的行为方式还不可知。

653

找出谁持有锁

如果你看到有许多进程都处于 Locked 状态，那你的问题可能是将 MyISAM 或类似的存储引擎用于高并发的 workload 上了。这会让你无法执行人工操作，例如往表里增加一个索引。如果一个 UPDATE 查询被放入队列里，等待着一个 MyISAM 表的锁，那就连一个 SELECT 查询也不会被允许运行。（你在 MySQL 使用手册里可以读到更多关于 MySQL 锁队列和优先级的内容。）

在一些案例里，你会清晰地看到有几个连接长时间地持有着某个锁，你要做的就是把它们杀掉就行了（或者是劝告某个用户不要耽搁这些工作的运行）。但问题是你怎样才能找出那些连接呢？

现在，没有 SQL 命令可以用来显示那些持有着表锁从而阻塞了你查询的线程。如果你运行 SHOW PROCESS LIST，就能看到在等待锁的进程，但是看不出持有着锁的进程。幸运的是，有一个 debug 命令可供使用（它不可能通过 SQL 来运行），它能够将锁的信息打印到服务器的错误日志里。你可以使用 mysqladmin 辅助功能来运行这个命令：

```
$ mysqladmin debug
```

输出内容里包含了大量的调试信息，但是，在接近末尾的地方，你会看到类似下面这样的信息。在输出这些信

息时，我们在一个连接里锁定了表，然后在另一个连接里试着再去锁定它：

```
Thread database.table_name Locked/Waiting Lock_type
7      sakila.film        Locked - read   Read lock without concurrent inserts
8      sakila.film        Waiting - write Highest priority write lock
```

在此，你能看到线程 8 正等待着线程 7 持有的锁。

如果 MySQL 是在调试开启的状态下编译出来的，mysqladmin debug 还将输出更多的信息，但是，关于锁和与此相关的其他有用信息也就这么一些。

654

D.1.2 全局读锁

The Global Read Lock

MySQL 服务器也实现了一个全局读锁。你可以像下面这样获得这个锁：

```
mysql> FLUSH TABLES WITH READ LOCK;
```

如果你现在在另一个会话里试着再去锁定这个表，它就会被挂起：

```
mysql> LOCK TABLES sakila.film WRITE;
```

你如何才能分辨出这个查询等待的是全局读锁，而不是像以前那样的表级锁呢？请看 SHOW PROCESS LIST 的输出：

```
mysql> SHOW PROCESSLIST\G
...
***** 2. row *****
      Id: 22
      User: baron
      Host: localhost
      db: NULL
      Command: Query
      Time: 9
      State: Waiting for release of readlock
      Info: LOCK TABLES sakila.film WRITE
```

请注意这个查询的状态是 Waiting 一个死锁的释放。这就说明该查询等待的是全局读锁，不是表级锁。

目前，MySQL 还没有方法帮你找出谁持有着这个全局读锁。

D.1.3 名称锁

Name Locks

名称锁是表锁的一种，在服务器重命名或删除一个表时创建。名称锁会跟一个普通的表锁冲突，无论是显式的，还是隐含的。举例来说，如果你先使用了 LOCK TABLES，然后，另外一个会话试着去重命名这个锁定的表，这时，这个查询就会被挂起，但这次不是处于 Locked 状态：

```
mysql> RENAME TABLE sakila.film2 TO sakila.film;
```

如同前文一样，进程列表是用来查看被锁定的查询——它们的表状态都是 Waiting：

```
mysql> SHOW PROCESSLIST\G
...
***** 2. row *****
      Id: 27
```

```

User: baron
Host: localhost
db: NULL
Command: Query
Time: 3
State: Waiting for table
Info: rename table sakila.film to sakila.film 2

```

你也可以通过 `SHOW OPEN TABLES` 的输出信息来查看这个效果：

```

mysql> SHOW OPEN TABLES;
+-----+-----+-----+-----+
| Database | Table      | In_use | Name_locked |
+-----+-----+-----+-----+
| sakila   | film_text  | 3      | 0           |
| sakila   | film       | 2      | 1           |
| sakila   | film2      | 1      | 1           |
+-----+-----+-----+-----+
3 rows in set (0.00 sec)

```

请注意那两个名称（原来的和新改的）都被锁定了。`sakila.film` 里的一个触发器引用了 `sakila.film_text`，因此，它也被锁定了，这说明锁会通过另外一条迂回的路径，出乎你意料地发生作用。如果你去查询 `sakila.film`，触发器会促使你在背后接触到 `sakila.film`，从而将它隐含地锁定。触发器确实无须因为重命名而触发，因而在技术上也不需要请求锁。但是，事实是：MySQL 锁的粒度有时没你希望的那么恰如其分。

MySQL 也没有提供任何找出谁持有着名称锁的方法，但是，这通常不构成问题，因为它们一般只会被持有很短的一段时间。当有冲突发生时，其原因往往是一个名称锁在等待另一个普通的表锁，这样，你就可以使用前文中用过的 `mysqladmin debug` 来显示它们了。

D.1.4 用户锁

User Locks

服务器上的最后一种锁是用户锁，它是基于命名互斥量来实现的。你可以指定一个字符串，然后试着去锁定，并在超时之前获取到它的锁：

```

mysql> SELECT GET_LOCK('my lock', 100);
+-----+
| GET_LOCK('my lock', 100) |
+-----+
| 1 |
+-----+
1 row in set (0.00 sec)

```

如果这个尝试立即成功地返回了，那么，现在这个线程就在那个命名互斥量上获得了锁。如果还有另外的线程试着去锁定同一个字符串，它就会被挂起直到超时。这一次，进程列表会显示出不同的状态信息：

```

mysql> SHOW PROCESSLIST\G
***** 1. row *****
    Id: 22
   User: baron
  Host: localhost
    db: NULL
Command: Query
   Time: 9

```



```
State: User lock
Info: SELECT GET_LOCK('my lock', 100)
```

User lock 状态是这种类型的锁独有的。MySQL 没有提供找出谁持有了某个用户锁的方法。

D.2 存储引擎里的锁等待

Lock Waits in Storage Engines

服务器级锁调试起来要比存储引擎里的锁容易一点点。存储引擎的锁会因存储引擎的不同而各不相同，而且，存储引擎也没有提供任何可用于检查锁的方法。在本附录里，我们主要以 InnoDB 为例，因为它是目前使用最广泛的实现了锁的存储引擎。

D.2.1 InnoDB 锁等待

InnoDB Lock Waits

InnoDB 在 SHOW INNODB STATUS 里显露出了一些锁的信息。如果一个事务正在等一个锁，那么这个锁就会出现在 SHOW INNODB STATUS 输出信息的 TRANSACTIONS 段里。举例来说，如果你在一个会话里执行下面这个命令，你就能在表的第 1 行上取得一个读锁：

```
mysql> SET AUTOCOMMIT=0;
mysql> BEGIN;
mysql> SELECT film_id FROM sakila.film LIMIT 1 FOR UPDATE;
```

如果现在你在另一个会话里执行同样的命令，你的查询就会因刚才那行的读锁而阻塞。通过 SHOW INNODB STATUS，你可以达到这种效果（出于表述清晰的目的，我们对输入结果做了一些删减）：

```
1 LOCK WAIT 2 lock struct(s), heap size 1216
2 MySQL thread id 8, query id 89 localhost baron Sending data
3 SELECT film_id FROM sakila.film LIMIT 1 FOR UPDATE
4 ----- TRX HAS BEEN WAITING 9 SEC FOR THIS LOCK TO BE GRANTED:
5 RECORD LOCKS space id 0 page no 194 n bits 1072 index `idx_fk_language_id` of table
   `sakila/film` trx id 0 61714 lock_mode X waiting
```

最后一行显示出该查询正在等待一个位于该表 idx_fk_language_id 索引第 194 页上的独占锁(lock_mode X)。最终以等待超时而结束，该查询将返回一个报错信息：

```
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
```

不幸的是，由于没有办法查看到是谁持有着锁，也就很难找出到底是哪个事务引发了这个问题。你也可以通过查看哪个事务打开了很长的一段时间来作一个有根据的推测；或者，你也可以激活 InnoDB 锁监控器，它可以显示出每一个事务持有的 10 个锁。为了激活这个监控器，你有必要使用 InnoDB 存储引擎创建一个神奇的命名表（注 2）：

```
mysql> CREATE TABLE innodb_lock_monitor(a int) ENGINE=INNODB;
```

当你发出这个查询时，InnoDB 会开始定时（间隔时间各有不同，但是通常都是每分钟几次）地把内容略有增强的 SHOW INNODB STATUS 输出信息打印到标准输入上。在多数系统里，这些输出信息被重定向到服务器的错误日志里，你就可以检查日志信息，来看看哪个事务持有着那些锁。要停止锁监控，删除这个表就可以了。

注 2：InnoDB 把几个“神奇的”表名作为操作指令来用。当前采用的是使用动态可设置的服务器变量，但是，InnoDB 的方法已经使用了很长一段时间，所以，仍然还留有一些原先的行为方式。

以下是锁监控器输出的相关样例：

```
1 ---TRANSACTION 0 61717, ACTIVE 3 sec, process no 5102, OS thread id 1141152080
2 3 lock struct(s), heap size 1216
3 MySQL thread id 11, query id 108 localhost baron
4 show innodb status
5 TABLE LOCK table `sakila/film` trx id 0 61717 lock mode IX
6 RECORD LOCKS space id 0 page no 194 n bits 1072 index `idx_fk_language_id` of table
  `sakila/film` trx id 0 61717 lock_mode X
7 Record lock, heap no 2 PHYSICAL RECORD: n_fields 2; compact format; info bits 0
8 ... 省略 ...
9
10 RECORD LOCKS space id 0 page no 231 n bits 168 index `PRIMARY` of table `sakila/film`
   trx id 0 61717 lock_mode X locks rec but not gap
11 Record lock, heap no 2 PHYSICAL RECORD: n_fields 15; compact format; info bits 0
12 ... 省略...
```

请注意第 3 行显示的 MySQL 线程 ID，它跟进程列表里 `Id` 列的值是一样的。第 5 行显示了该事务在表里有一个显式的独占表锁 (IX)。第 6 到 8 行显示了索引里的锁。我们删除了第 8 行的信息，是因为它导出了这个锁定的记录，显得非常累赘。第 9 到 11 行显示了主键上相应的锁 (FOR UPDATE 锁必须锁住整行，而不仅仅是索引)。

有一点在文档里没有提到，就是当锁监控器激活的时候，SHOW INNODB STATUS 里也会出现额外的信息，因此，你实际上无须特意到服务器的错误日志里查看锁信息。

更多有用的锁输出

出于种种原因，锁监控器是没经过优化的。其中的主要问题是锁信息非常冗长，它包含了被锁定记录的十六进制格式和 ASCII 格式导出。它会填满错误日志，还会在固定长度的 SHOW INNODB STATUS 输出结果里很轻易地溢出。这意味着你无法查看到在那一段之后的其他输出信息了（更多内容请查看第 569 页的“LATEST DETECTED DEADLOCK”）。InnoDB 也有一个硬编码在里面的限制，就是每个事务只能打印出 10 个持有的锁，超过 10 个的，就不打印出来了，这意味着你无法看到你想要看的锁的任何信息。如果想要把它提到最上面去，即使你要找的东西确实在里面，你也难以把它从所有的锁的输出信息里找出来。（若在一个繁忙的系统上试一下，你就会体会到这一点。）

有两样东西能够使锁的输出信息更加有用。第一样是本书作者之一为 InnoDB 和 MySQL 服务器编写的一个补丁。这个补丁会移除输出结果里那些冗长的记录导出信息，默认情况下会把锁信息包含到 SHOW INNODB STATUS 的输出结果里（这样，锁监控器就无须激活了），还会增加动态可设置服务器变量来控制冗长的输出信息以及每个事务能打印出的锁信息的个数。你可以在 <http://lists.mysql.com/internals/35174> 上获取用于 MySQL 5.0 的补丁。

第 2 个可选用的方法是使用 innotop 来解析和格式化输出结果。它的 Lock 模式能够显示锁信息，并通过连接和表优美地聚合在一起，这样你就能很快地看出哪一个事务持有者指定表上的锁。但是，这也并非是万无一失的方法，因为它是通过检查所有被锁定记录的导出信息找出那个精确的被锁定记录的。无论怎样，这还是要比常用的方法好很多，在许多用途里都足够好了。

InnoDB 开发者告诉过我们：他们正在把 InnoDB 信息导出到 INFORMATION_SCHEMA 表里，会在将来的某个版本里发布，但是，其中的代码还没公布出来。在将来，这可能就是显示锁信息的最佳方法。

658

D.2.2 Falcon 锁等待

Falcon Lock Wait

在本书写作的时候，Falcon 事务存储引擎已是 MySQL 6.0 alpha 版的一部分，它将事务信息导出到一个 INFORMATION_SCHEMA 表里。于是，你可以使用 SQL 命令很方便地找出引发锁等待的起因：

```
mysql> SELECT a.THREAD_ID AS blocker, a.STATEMENT AS blocking_query,
->      b.THREAD_ID AS blocked, b.STATEMENT AS blocked_query
-> FROM INFORMATION_SCHEMA.FALCON_TRANSACTIONS AS a
->      INNER JOIN INFORMATION_SCHEMA.FALCON_TRANSACTIONS AS b ON
->      a.ID = b.WAITING_FOR
-> WHERE b.WAITING_FOR > 0;
```

| blocker | blocking_query | blocked | blocked_query |
|---------|----------------|---------|------------------------------|
| 4 | | 5 | SELECT * FROM tbl FOR UPDATE |

在将来，对于 MySQL 数据库管理员而言，这种类型的诊断信息会使生活变得更加容易。

Symbols

- * (asterisk), passwords beginning with, 529
- % (percent sign), default hostname, 526, 534
- ? (question mark), parameters in prepared statements, 225
- '...' (quotes), for hostnames and usernames, 535

A

- ab tool, 42
- Aborted_clients status variable, 301, 559
- Aborted_connects status variable, 301, 559
- access control, 522
- access time, of hard disk, 315, 316
- accounts, 522
 - adding, 525, 526
 - anonymous, disabling, 534
 - privileges associated with displaying, 525
 - for running MySQL, 541
 - types of, 522
 - removing, 525
 - for replication, creating, 347
 - stored in grant tables, 523
 - types of, 526
- ACID test, 7
- active caches, 463
- active monitoring, 585
- adaptive hash indexes, 103, 575
- administration, software for, 584, 585
- administrator account
 - database, 527
 - system, 527
- AES_DECRYPT() function, 554
- AES_ENCRYPT() function, 554
- Aker, Brian (functions for memcached), 230
- algebraic equivalence rules, for query optimizations, 167
- ALTER TABLE command, 138
 - improving performance of, 145–148
 - table conversions using, 30
- analysis tools, 595–598
- ANALYZE TABLE command, 137
- Analyzing state, of query, 164
- anonymous users, disabling, 534
- Apache web server, 460–462
- application-level caching, 465–467
- application-level encryption, 552–554
- applications
 - joins performed in, 159
 - performance of
 - caching for, 463–469
 - extending MySQL for, 470
 - optimal concurrency, finding, 462
 - problems with, finding, 457–460
 - web server problems, 460–463
 - profiling, 55–63, 457
- Archive storage engine, 21, 29
- archiving data
 - for scalability, 432–435
 - replication for, 372
 - utility for, 599
- asterisk (*), passwords beginning with, 529
- Atomicity, in ACID test, 7

欢迎大家提出建议来改善索引内容。建议请发 E-mail 到 index@oreilly.com。

- attributes support, in Sphinx, 638
- auditing, backups for, 477
- authentication, 3, 521
- authorization, 521
 - (see also privileges)
- auto_increment_increment variable, 364
- auto_increment_offset variable, 364
- AUTOCOMMIT mode, 11
- autogenerated schemas, 96
- automatic host blocking, 549
- availability, high, 409, 410, 447–456
 - adding redundancy, 449
 - failover and failback for, 452–456
 - IP addresses, moving, 453
 - IP takeover for, 453
 - masters, switching roles of, 453
 - middleman solutions for, 454
 - MySQL Master-Master Replication
 - Manager tool for, 454
 - planning for, 447
 - replicated-disk architectures for, 449
 - replication for, 345
 - shared-storage architectures for, 449
 - slave, promoting, 453
 - synchronous replication for, 451

B

- Background Patrol Read, 320
- Backup & Recovery* (Preston), 472
- backup accounts, 528
- BACKUP DATABASE command, 517
- backups, 472–477
 - for auditing, 477
 - of binary logs, 476, 486–488
 - choosing storage engine based on, 25
 - “cold” backups, 473
 - copying files between machines, 476
 - data and file consistency of, 483–485
 - delimited file backups, 490
 - for disaster recovery, 476
 - filesystem snapshots, 492–499
 - “hot” backups, 473
 - importance of, 472, 476
 - incremental, 482
 - location of, 542
 - logical backups, 476, 479
 - creating, 489–492, 511, 517
 - restoring, 502–504
 - LVM snapshots, 492–499
 - monitoring, 476

- offline, 478
- online, 478
- parallel dump and restore, 491
- RAID as, 475
- raw backups, 476, 480, 500
- recommendations for, 475
- replication for, 345, 475, 485
- replication slave initialization using, 353
- scripting, 518–520
- security of, 476
- by shared hosting provider, 477
- snapshot-based backups, 476
- speed of, 510
- SQL dumps, 489
- testing, 476, 481, 504
- for testing purposes, 477
- tools for, 511–517, 600
- “warm” backups, 473
- what data to include in, 477, 481–483
- (see also recovery)

Barth, Wolfgang (*Nagios System and Network Monitoring*), 587

battery backup unit (BBU), 324, 327

BBU (battery backup unit), 324, 327

BENCHMARK() function, 45

benchmarking, 32

- automating, 41
- before configuring server, 270
- changing parameters of, 40
- common mistakes in, avoiding, 37
- concurrency measurements, 36
- data set for, 38
- designing benchmarks, 38
- errors during, 38
- examples of, 44–54
- full-stack, 33, 42
- goals for, determining, 34–37
- interference by other jobs, 40
- latency, 35
- of migrations, 40
- number of times to run, 41
- queries for, 39
- realistic scenarios for, 37
- reasons for, 33
- repeatability of, 39, 40
- results of
 - accuracy of, 39
 - analyzing, 41
 - documenting, 39
 - unusual, 41
- scalability measurements, 35

single-component, 33, 34, 43
 standard benchmarks, when to use, 38
 tools for, 42–45
 ab tool, 42
 BENCHMARK() function, 45
 Database Test Suite, 43, 51–53
 http_load tool, 42, 44
 JMeter, 42
 MySQL Benchmark Suite
 (sql-bench), 43, 53
 mysqslap tool, 43
 Super Smack, 44
 sysbench tool, 43, 46–50
 transactions per time unit
 (throughput), 34
 warming up system before, 38
 BIGINT type, 82
 binary log events, replication, 345
 binary logs
 backing up, 476, 486–488
 flushing, InnoDB, 294
 format of, 487
 purging, 488
 for replication, 345, 354
 status of, 581
 status variables for, 559
 bind_address variable, 543
 binlog dump process, 346
 Binlog_cache_disk_use status variable, 301, 559
 Binlog_cache_use status variable, 301, 559
 binlog_do_db variable, 360
 binlog_ignore_db variable, 360, 373
 BIT type, 91
 bit-packed data types, 91–93
 Blackhole storage engine, 22, 29
 “blob streaming” infrastructure, PBXT, 24
 BLOB types, 86, 298–301
 books and publications
 Backup & Recovery (Preston), 472
 Building Internet Firewalls (Zwicky et al.), 542
 High Performance Web Sites (Souders), 461
 MySQL documentation, 559
 MySQL Stored Procedure Programming (Harrison, Feuerstein), 217
 Nagios System and Network Monitoring (Barth), 587
 Practical Unix and Internet Security (Garfinkel et al.), 541

TCP/IP Network Administration
 (Hunt), 542
 Boolean full-text searches, 247
 B-Tree indexes, 97–101
 limitations of, 100
 when to use, 99
 buffer pool, 271, 277
 BUFFER POOL AND MEMORY section,
 SHOW INNODB STATUS
 output, 576
 buffer pool, status of, 576
Building Internet Firewalls (Zwicky et al.), 542
 bulletin boards, storage engines suited for, 28
 Bytes_received status variable, 302
 Bytes_sent status variable, 302

C

CACHE INDEX command, 275
 cache miss rate, 314
 cache tables, 142–145
 cache units, 313
 caches, CPU, 309
 caching, 463–469
 active caches, 463
 application-level, 465–467
 below application level, 464
 control policies for, 467
 memory requirements for, 274–281
 object hierarchies for, 468
 passive caches, 463
 pregenerating content for, 469
 reads and writes affected by, 311
 (see also specific caches)
 caching proxy server, 461
 Cacti tool, 329, 590
 capacity, 410
 CD-ROM applications, storage engines
 suited for, 28
 certificates, SSL (see SSL)
 CHANGE MASTER TO command, 349, 352, 382, 383
 CHAR type, 84–86
 CHAR_LENGTH() function, 242
 CHARACTER SET clause, 239
 character sets, 237–240
 character length affected by, 242
 choosing, 240, 243
 client/server communication of, 238
 comparison of values between, 239

- character sets (*continued*)
 - default, 237
 - effects on queries, 241–244
 - escape sequences, handling of, 240
 - index limitations affected by, 242
 - specifying in statements, 239, 240
 - supported, determining, 240
- character_set_client variable, 238
- character_set_connection variable, 238
- character_set_database variable, 239
- character_set_result variable, 238
- CHECK TABLE command, 136
- checksum queries, 600
- chrooted environment, MySQL in, 554
- CIPHER requirement option, 547
- client/server protocol, 161–164
- clustered indexes, 110–119
 - advantages of, 112
 - disadvantages of, 112
 - InnoDB implementation of, 19, 111, 113–119
- clustered systems, 435
- code reuse, 217
- coercibility of values, 239
- “cold” backups, 473
- Cole, Jeremy (SHOW PROFILE patch), 74
- collate clauses, 239
- collations, 237–240
 - choosing, 240
 - client/server communication of, 238
 - default, 237
 - effects on queries, 241–244
 - escape sequences, handling of, 240
 - specifying in statements, 239, 240
 - supported, determining, 240
- column privileges, query cache not used with, 532
- columns_priv table, 524
- Com_* status variables, 302
- Com_admin_commands status variable, 560
- Com_change_db status variable, 560
- Com_select status variable, 209, 210, 560
- command counters, 560
- command-line scripting, noninteractive mode for, 591
- comments in stored code, 224
- COMMIT command, 6
 - (see also AUTOCOMMIT mode)
- compressing files, 603–606
- concurrency, 3–6
 - choosing storage engine based on, 25
 - internal concurrency issues, 309
 - locks determining level of, 14
 - logical concurrency issues, 308
 - MVCC (multiversion concurrency control), 12–14, 150
 - optimal, 462
 - tuning, 295–297
 - (see also locks)
- concurrency measurements, 36
- concurrent_insert variable, 296
- configuration of server (see server configuration)
- connection pooling, compared to persistent connections, 460
- CONNECTION_ID() function, caching not used for, 205
- connections, 2
 - after removing all privileges, 537
 - authentication for, 3, 521
 - encryption for, 545–548
 - errors from, 533
 - list of, 578
 - localhost, Unix socket used for, 533
 - localhost-only connections, 543
 - memory required for, 273
 - per-connection settings, tuning, 304
 - status variables for, 559
 - troubleshooting, 76–79, 533
- Connections status variable, 302, 558
- consistency of data, with backups, 484
- Consistency, in ACID test, 7
- constant expressions, optimizations for, 167
- Continuous Data Protection, 517
- CONVERT() function, 239
- copying files
 - benchmarks for, 606
 - large files, 603–606
- Copying to tmp table state, of query, 164
- correlated subqueries, optimization of, 179–183
- COUNT() function, optimizations for, 167, 188–190
- counter tables, 144
- covering indexes, 120–124, 168
- CPU benchmark, sysbench tool, 46
- CPU-bound server, 339
- CPUs

- architecture of, 308
- number of, 306–309, 414
- profiling usage of, tools for, 63
- saturation of, 306–309
- speed of, 306–308

- crash recovery, choosing storage engine
 - based on, 26

- CREATE TEMPORARY TABLE
 - command, 21

- CREATE USER privilege, 537

- Created_tmp* status variables, 560

- Created_tmp_disk_tables status
 - variable, 302

- Created_tmp_tables* status variable, 302

- Cricketer tool, 590

- CSV storage engine, 21, 29

- current waits, status of, 566

- CURRENT_DATE() function, caching not
 - used for, 205

- CURRENT_USER() function, caching not
 - used for, 205

- cursors, 224

D

- data archiving (see archiving data)

- data consistency, with backups, 484

- data dictionary, 280

- data distribution, replication for, 344

- data encryption (see encryption)

- data files (see files)

- data fragmentation, 138

- data sharding, 417–419

- aggregating data with Sphinx, 634, 648

- arranging shards on nodes, 423

- dynamic allocation of data to shards, 426

- explicit allocation of data to shards, 427

- fixed allocation of data to shards, 424

- fixed and dynamic allocation, 427

- globally unique IDs needed for, 429–431

- partitioning function for, 424

- partitioning keys for, 419–421

- querying across shards, 421

- rebalancing shards, 428

- size of shards, 422

- in Sphinx, 636

- time-based data partitioning, 434

- tools for, 431

- unit of sharding for, 420

- data synchronization (see replication)

- data types

- autogenerated schemas choosing, 96

- optimal, 80

- bit-packed, 91–93

- choosing, 81

- date and time, 82, 90

- for identifier columns, 93

- nullable, 81

- real numbers, 83

- size of, 81, 86

- strings, 84–90

- whole numbers, 82

- database

- files for, 326–328

- host for, restricting logins on, 541

- migrating to MySQL, 584

- privileges for, 535

- storage location of, 14

- database administrator accounts, 527

- Database Test Suite, 43, 51–53

- date and time data types, 90

- high-resolution support, 91

- optimal, choosing, 82

- DATETIME type, 82, 90

- db table, 523

- dbt2 tool, Database Test Suite, 51–53

- deadlocks, 9

- monitoring, 598

- status of, 569–571

- debug command, mysqladmin, 653

- debugging (see troubleshooting)

- DECIMAL type, 83

- decompressing files, 603–606

- DEFAULT keyword

- for configuration variables, 268

- for MyISAM recovery, 282

- default route, not having, for firewall, 544

- DELAY_KEY_WRITE option, 18

- delay_key_write variable, 281, 296

- delayed key writes, MyISAM, 18

- DELAYED option, 196

- delayed replication, for recovery, 506

- Delayed_* status variables, 564

- DELETE command, EXPLAIN command
 - with, 609

- delimited file backups, 490, 503

- denormalization, 139, 140–142

- DETERMINISTIC option, 219

- development, software for, 584

- directio() function, 289

dirty read, 8
 disaster recovery, 476
 discussion forums, storage engines suited for, 28
 disks (see hard disks)
 DISTINCT clause, optimizing, 191
 distributed memory caches, 466
 distributed (XA) transactions, 262, 564
 distribution master, 369
 distribution of data, replication for, 344
 DMZs, 545
 DNS, performance of, avoiding reliance on, 329
 document pointers, 244
 documentation for MySQL, 559
 Dormando's Proxy for MySQL tool, 599
 double buffering, 313
 with fsync(), 288
 with O_SYNC flag, 289
 DOUBLE type, 83
 doublewrite buffer, InnoDB, 293
 DRBD, disk replication tool, 449
 drop command, mysqladmin, 540
 DROP USER command, 525
 duplicate indexes, 127–129
 Durability, in ACID test, 7
 dynamic optimizations, for queries, 166

E

edge side includes (ESI), 461
 employee accounts, 527
 ENCRYPT() function, 550
 encryption
 application-level, 552–554
 of connections, 545–548
 of data, 550–554
 of filesystems, 551
 hashing passwords, 529, 550
 in MySQL, 554
 engines (see storage engines)
 ENUM type, 87, 94
 equality propagation, 169, 184
 escape sequences, 240
 ESI (edge side includes), 461
 event counters, 566
 events, 222
 binary log events, 345
 (see also stored code)
 exclusive locks (write locks), 4
 execution plan of query, 172, 178
 (see also EXPLAIN command)

expire_logs_days variable, 295, 358, 488
 EXPLAIN command
 for non-SELECT queries, 609
 invoking, 607
 mk-visual-explain script for, 621
 output from, 607, 610–621
 extra column, 621
 filtered column, 620
 id column, 610
 key column, 617
 key_len column, 617
 partitions column, 610
 possible_keys column, 617
 ref column, 618
 rows column, 619
 select_type column, 612
 table column, 612–615
 in tree structure, 621
 type column, 615–617
 performance of, 608
 EXPLAIN EXTENDED command, 608, 620
 EXPLAIN PARTITIONS command, 608, 610
 explicit invalidation, cache control policy, 467
 explicit locking, 12
 ext2 filesystem, 332, 333
 ext3 filesystem, 332, 333
 extended command, mysqladmin, 70, 276, 558, 559, 564
 external XA transactions, 264

F

failback, 452–456
 failover, 345, 452–456
 Falcon storage engine, 23, 29
 integer types, storage of, 82
 lock waits, 658
 locking in, 14
 MVCC supported by, 12
 fastest response, load-balancing algorithm, 444
 fault tolerance, 410, 411
 fdatsync() function, 288
 Federated storage engine, 21, 29, 436
 federation, 412, 436
 Feuerstein, Steven (*MySQL Stored Procedure Programming*), 217
 file consistency, with backups, 484
 file descriptors, status variables for, 561

FILE I/O section, SHOW INNODB STATUS
 output, 574

FILE privilege, 528

fileio benchmark, sysbench tool, 47

files

- compressing and decompressing, 603–606
- copying large files, 603–606
- copying, benchmarks for, 606
- database, 326–328
- reading and flushing, 288–290
- server configuration, 266, 267

filesorts, optimizing, 176, 300

filesystem snapshots, 492–499
 (see also LVM)

filesystems

- choosing, 331–333
- encryption of, 551

filtering, in Sphinx, 639

firewalls, 544

FLOAT type, 83

FLUSH HOSTS command, 549

FLUSH PRIVILEGES command, 526

FLUSH QUERY CACHE command, 213

FLUSH STATUS command, 71

FLUSH TABLES WITH READ LOCK
 command, 478, 653

flush-hosts command, mysqladmin, 549

FNV (Fowler/Noll/Vo) UDF, 105

FOR UPDATE option, 197

FORCE INDEX option, 197

foreign keys, 20, 150, 252

- errors in, 567–569
- redundant, checking for, 597

Fowler/Noll/Vo (FNV) UDF, 105

fragmentation of data, 138

FreeBSD operating system, 331

.frm files, 14, 146

fsync() function, 288, 565

ft_min_word_len parameter, 251

full-stack benchmarking, 33, 42

full-text searching, 244–248

- Boolean, 247
- changes in version 5.1, 248
- collection for, 244
- indexes for, 106
- limitations of, 249–251
- natural-language, 245–247
- parser plug-ins for, 470
- replication slaves used for, 373
- with Sphinx, 627, 643–645

- tuning and optimization, 251–252

functional partitioning, 416, 446

functions

- nondeterministic, caching not used for, 205
- stored, 219
- user-defined (UDFs), 230, 470

G

Galbraith, Patrick (functions for memcached), 230

Garfinkel, Simson (*Practical Unix and Internet Security*), 541

gdb tool, 78

general query log, 64

GET_LOCK() function, 655

global locks, 651

global privileges, 522, 537

global read locks, 653

global version/session split, 440

GNU/Linux operating system, 330, 331

gprof tool, 79

GRANT command, 525, 526

- analyzing, 600
- not replicating, 361

grant tables, 523

- how MySQL uses, 524
- modifying directly, 525, 526

granularity of locks, 4

Grimmer, Lenz (mylvmbackup tool), 492, 515

Groundwork Open Source tool, 588

GROUP BY clause, optimizing, 191, 630, 646

group commit, 263

groups, simulated, 528

gunzip tool, 604, 606

gzip compression, enabling, 461

gzip tool, 604, 605, 606

H

HackMySQL tools, 595

handler operations, status variables for, 560

Handler_* status variables, 560

Handler_read_rnd_next status variable, 302

hard disks

- choosing, 315–317
- memory-to-disk ratio for, 314
- multiple disk volumes, 326–328
 (see also RAID)

hardware
 for slave server, 317
 upgrading, 414
 (see also CPUs)
 Harrison, Guy (*MySQL Stored Procedure Programming*), 217
 hash code, 101
 hash functions, 101, 103, 104, 550
 hash indexes, 101–106, 149
 adaptive hash indexes, 103, 575
 collisions, handling, 105
 emulating, 103
 limitations of, 102
 hash joins, emulating, 185
 hashed, load-balancing algorithm, 444
 hashing passwords, 529, 550
 have_openssl variable, 546
 HEAP tables (see Memory storage engine)
 heartbeat record, 379
 helper threads, status of, 574
 HFS Plus filesystem, 333
 Hibernate Shards, 432
 hidden privileges, 537–540
 high availability (see availability, high)
 High Availability Linux project, 452
High Performance Web Sites (Souders), 461
 HIGH_PRIORITY option, 195
 HiveDB, 432
 host blocking, automatic, 549
 host table, 524
 hostnames
 default, 526
 localhost, 533
 quoting in commands, 535
 “hot” backups, 473
 http_load tool, 42, 44
 Hunt, Craig (*TCP/IP Network Administration*), 542
 Hyperic HQ tool, 588

I

ibbackup (InnoDB Hot Backup tool), 513, 517
 .ibd files, 291
 identifier columns, data types for, 93
 ifconfig tool, 336
 IGNORE INDEX option, 197
 implicit locking, 12
 IN() list comparisons, optimizations of, 169
 incremental backups, 482
 index merge algorithms, optimization of, 183
 index scans, loose, not supported, 185
 index statistics, 170
 index writes, deferring, 281
 index-covered queries, 121
 indexer program, in Sphinx, 635
 indexes, 95, 130
 B-Tree indexes, 97–101
 limitations of, 100
 when to use, 99
 corruption of, 136
 covering indexes, 120–124, 168
 example of, 131–135
 fragmentation of, 138
 full-text indexes, 106
 hash indexes, 101–106
 adaptive hash indexes, 103, 575
 collisions, handling, 105
 emulating, 103
 limitations of, 102
 inserting records in primary key order, 117–120
 in MyISAM, 18
 performance of
 building quickly, 148
 clustered indexes, 110–119
 covering indexes, 120–124
 duplicate indexes, 127–129
 index scans for sorts, 124–126
 isolating columns, 106
 locking and, 129–131
 packed indexes, 126
 prefix indexes, 107–110
 redundant indexes, 127–129, 597
 sorting, 135
 range conditions and, 134
 selectivity of, 107–110
 spatial indexes, 106
 statistics for, updating, 136
 surrogate keys, 117
 indexing, full-text (see full-text searching)
 info() function, 136
 INFORMATION_SCHEMA database, 558, 581
 finding obsolete privileges with, 540
 system variable access in, 557
 table privileges for, 532
 InnoDB Hot Backup tool (ibbackup), 513, 517
 InnoDB Recovery Toolkit, 510

- InnoDB storage engine, 19, 29
 - adaptive hash indexes, 103
 - AUTO_INCREMENT, table locking used for, 151
 - buffer pool, 271, 277
 - clustered indexes, 19, 111, 113–119, 150
 - concurrency tuning, 19, 296
 - corruption problems with, 508
 - COUNT(*) queries not optimized, 151
 - data dictionary, 280
 - data loading, not optimized, 151
 - filesystem snapshots for, 497
 - foreign key constraints in, 20, 150
 - I/O, tuning, 283–295
 - binary log settings, 294
 - doublewrite buffer settings, 293
 - log file and buffer settings, 285–290
 - tablespace settings, 290–293
 - transaction log settings, 284–285
 - isolation levels in, 11, 19
 - lock waits in, 656–658
 - lock-free backups, 496–498
 - locking in, 12, 14, 19
 - monitoring, 591
 - MVCC supported by, 12, 13, 150
 - optimized caching, 150
 - recovery for, 507–510
 - redundant indexes, 128
 - restoring raw files, 500
 - row locks, 150
 - 64-bit numbers, output format for, 565
 - status information for, 565–578
 - adaptive hash index, 575
 - buffer pool, 576
 - current waits, 566
 - deadlocks, 569–571
 - event counters, 566
 - foreign key errors, 567–569
 - I/O helper threads, 574
 - insert buffer, 575
 - mutexes, 579
 - performance counters, 574
 - row operations, 577
 - transaction logs, 576
 - transactions, 572–574
 - status variables for, 563
 - tablespaces in, 19
 - transactions affecting query cache, 206, 215
 - unpacked indexes, 150
- InnoDB_* status variables, 563
- InnoDB_buffer_pool_pages_dirty status variable, 278
- innodb_buffer_pool_size variable, 271
- innodb_commit_concurrency variable, 297
- innodb_concurrency_tickets variable, 297
- innodb_data_file_path variable, 291
- innodb_data_home_dir variable, 291
- innodb_doublewrite variable, 294
- innodb_file_io_threads variable, 290
- innodb_file_per_table variable, 281, 291
- innodb_flush_log_at_trx_commit variable, 286
- innodb_flush_method variable, 288
- innodb_force_recovery variable, 510
- innodb_log_buffer_size variable, 286
- innodb_log_file_size variable, 271
- innodb_max_dirty_pages_pct variable, 278
- innodb_max_purge_lag variable, 293
- innodb_open_files variable, 281
- InnoDB_os_log_written status variable, 286
- innodb_thread_concurrency variable, 297
- innodb_thread_sleep_delay variable, 297
- innotop tool, 559, 565, 591–595
- Input/Output (see I/O)
- INSERT BUFFER AND ADAPTIVE HASH INDEX section, SHOW INNODB STATUS output, 575
- insert buffer, status of, 575
- INSERT command
 - DELAYED, status variables for, 564
 - EXPLAIN command with, 609
 - with SELECT, for table conversions, 30
- INT type, 82
- interactive monitoring tools, 591–595
- interface tools, 583–585
- internal concurrency issues, 309
- internal XA transactions, 263
- intra-row fragmentation, 138
- introducers, for character sets, 239
- invalidation on read, cache control policy, 467
- invisible privileges, 537–540
- I/O
 - caches affecting, 311
 - for InnoDB, tuning, 283–295
 - binary log settings, 294
 - doublewrite buffer settings, 293
 - log file and buffer settings, 285–290
 - tablespace settings, 290–293
 - transaction log settings, 284–285
 - for MyISAM, tuning, 281–283

I/O (continued)

- random, 310
- sequential, 310
- I/O merging, 312
- I/O saturation, 306
- I/O slave thread, 346
- I/O-bound server, 340
- iostat tool, 338–339
- IP addresses, storing, 95
- isolation levels, 8
 - with InnoDB, 11, 19
 - MVCC support for, 14
 - setting, 11
 - (see also locks)
- Isolation, in ACID test, 7

J

- JFS filesystem, 332, 333
- JMeter tool, 42
- join_buffer_size variable, 267
- joins
 - decomposition of, 159
 - execution strategy for, 170–172
 - optimizations for, 167, 173–176, 190
 - STRAIGHT_JOIN option, 196

K

- Keep-Alive configuration, Apache, 460, 462
- key block size, 276
- key caches (key buffers), 269, 274–277
- Key_* status variables, 561
- KEY_BLOCK_SIZE option, 277
- Key_blocks_used status variable, 302
- key_buffer_size variable, 268, 269, 274
- Key_reads status variable, 302
- keys (see indexes)

L

- Last_query_cost status variable, 564
- latency, 315
 - benchmarking, 35
 - low, fast CPUs for, 307
 - for networks, 328
- LATEST DETECTED DEADLOCK section,
 - SHOW INNODB STATUS
 - output, 569–571
- LATEST FOREIGN KEY ERROR section,
 - SHOW INNODB STATUS
 - output, 567–569

- least connections, load-balancing
 - algorithm, 444
- LENGTH() function, 242
- lighttpd, lightweight server, 461
- lightweight profiling, 57
- LIMIT clause, optimizing, 193, 194
- Linux operating system (see GNU/Linux operating system)
- Linux Virtual Server (LVS), 438
- LinuxThreads thread library, 334
- load, 411
- load balancing, 436–447
 - algorithms for, 444
 - application configuration for, 440
 - connecting directly with, 438–442
 - data partitioning for, 446
 - DNS names, changing, 441
 - filtering for, 446
 - functional partitioning for, 446
 - goals of, 436
 - IP addresses, moving, 441
 - master and multiple slaves for, 445
 - middleman solutions for, 442–445
 - replication for, 345, 438
 - servers, adding or removing, effects
 - on, 445
 - tools for, 438, 442
- LOAD DATA FROM MASTER
 - command, 353
- LOAD DATA INFILE command, 239, 490
- LOAD INDEX command, 275
- LOAD TABLE FROM MASTER
 - command, 353
- local caches, 465
- local shared-memory caches, 466
- localhost hostname, 533
- localhost-only connections, 543
- locality of reference, 310
- LOCK IN SHARE MODE option, 197
- LOCK TABLES command, 12, 281, 651
- lock waits
 - displaying list of, 591
 - server level, 650–655
 - in storage engine, 655–658
- Locked state, of query, 163
- locks
 - with Archive engine, 21
 - concurrency level of, 14
 - debugging, 650–658
 - explicit, 12
 - Falcon, 658

- features listed by storage engine, 29
 - global locks, 651
 - global read locks, 653
 - granularity of, 4
 - implicit, 12
 - indexes affecting, 129–131
 - InnoDB, 19, 656–658
 - Memory engine, 20
 - MyISAM engine, 18
 - name locks, 651, 654
 - performance of, 5
 - read locks, 4
 - row locks, 6, 14, 150
 - status variables for, 563
 - string locks, 651
 - table locks, 5, 149, 650, 651–653
 - concurrency level of, 14
 - as potential bottleneck, 149
 - status variables for, 563
 - user locks, 655
 - who is holding, determining, 653, 656
 - write locks, 4
 - log buffer
 - flushing, 286
 - size of, 286
 - log files
 - reading and flushing, 288–290
 - separating from data files, 327
 - size of, 285
 - LOG section, SHOW INNODB STATUS
 - output, 576
 - log server
 - for recovery, 506
 - replication for, 374–376
 - log_bin_trust_function_creators
 - variable, 219
 - log_queries_not_using_indexes variable, 64, 65
 - log_slave_updates variable, 349, 359, 487
 - logging
 - binary logs (see binary logs)
 - query logs, 64–70
 - relay log, 346
 - storage engines suited for, 26
 - transaction logging, 10, 284–285, 576
 - logging accounts, 528
 - logical backups, 476, 479
 - creating, 489–492, 511, 517
 - restoring, 502–504
 - logical concurrency issues, 308
 - logical reads, 311
 - logical unit numbers (LUNs), 325
 - Logical Volume Manager (see LVM snapshots)
 - logins, restricting, 541
 - long_query_time variable, 64, 65, 66
 - LONGBLOB type, 86
 - LONGTEXT type, 86
 - loose index scans, not supported, 185
 - LOW_PRIORITY option, 195
 - low_priority_updates variable, 296
 - lsf tool, 78
 - LUNs (logical unit numbers), 325
 - LVM (Logical Volume Manager)
 - snapshots, 492–499
 - configuration for, 493
 - creating, 494
 - initializing slave using, 353
 - lock-free InnoDB backups
 - using, 496–498
 - mounting, 495
 - for online backups, 496
 - planning for, 498
 - removing, 495
 - LVS (Linux Virtual Server), 438
- ## M
- Maatkit tools, 596, 599
 - (see also specific mk-* tools)
 - Maria storage engine, 24, 29
 - master topologies, for replication (see replication, topologies for)
 - master.info file, 357
 - MATCH AGAINST clause, 245
 - materialized views, 236
 - MAX() function, optimizations for, 167, 187
 - max_allowed_packet variable, 379, 404
 - max_connection_errors variable, 549
 - max_length_for_sort_data variable, 300
 - max_packet_size variable, 162
 - max_sort_length variable, 300
 - Max_used_connections status variable, 303
 - Maxia, Giuseppe
 - library of stored routines by, 217
 - Sandbox script by, 352
 - McCullagh, Paul (developer of PBXT), 23
 - MD5() function, 550
 - MEDIUMBLOB type, 86
 - MEDIUMINT type, 82

- MEDIUMTEXT type, 86
- memcached, 466
 - for globally-unique IDs, 429, 430
 - user-defined functions for, 230
- memlock variable, 336
- memory
 - amount available to MySQL, 272
 - balancing with disk resources, 309–317
 - cache requirements for, 274–281
 - operating system requirements for, 273
 - peak consumption of, 273
 - query cache use of, 206–209
 - server configuration for, 272–274
- memory benchmark, sysbench tool, 50
- memory mapping, 283
- Memory storage engine, 20, 29
 - data not persisted, 150
 - dynamic rows not supported, 149
 - hash indexes, 101, 149
 - index statistics not supported, 150
 - locking in, 14
 - on-disk temporary tables used by, 87
 - table locks, 149
- memory-to-disk ratio, 314–315
- MERGE algorithm, for views, 232
- Merge storage engine, 14, 19, 29
- merge tables, 253–257
- migration of databases, software for, 584
- MIN() function, optimizations for, 167, 187
- mk-archiver tool, 373, 599
- mk-deadlock-logger tool, 598
- mk-duplicate-key-checker tool, 597
- mk-find tool, 599
- mk-heartbeat tool, 379, 598
- mk-parallel-dump tool, 492, 514, 517, 600
- mk-parallel-restore tool, 515, 600
- mk-profile-compact tool, 597
- mk-query-profiler tool, 596
- mk-show-grants tool, 600
- mk-slave-delay tool, 600
- mk-slave-prefetch tool, 600
- mk-slave-restart tool, 600
- mk-table-checksum tool, 380, 600
- mk-table-sync tool, 382, 600
- mk-visual-explain tool, 597, 621
- monitoring accounts, 528
- monitoring tools, 585–595
 - interactive tools, 591–595
 - noninteractive tools, 586–590
 - (see also status of MySQL server)
- MONyog tool, 589
- mpstat tool, 336
- MRTG (Multi Router Traffic Grapher), 329, 589
- mtop tool, 591
- Multi Router Traffic Grapher (MRTG), 329, 589
- multimaster replication, 364, 373
- multiple disk volumes, 326–328
- multivalued attributes (MVAs), 639
- multiversion concurrency control (see MVCC)
- Munin tool, 590
- mutex benchmark, sysbench tool, 50
- mutexes, status of, 579
- MVAs (multivalued attributes), 639
- MVCC (multiversion concurrency control), 12–14, 150
- my.cnf file, 266
- .MYD files, 16
- .MYI files, 16
- MyISAM storage engine, 16–19, 29
 - compact storage used by, 149
 - compressed tables in, 18
 - concurrency tuning, 295
 - COUNT() function, performance of, 189
 - data recovery not automated, 149
 - delayed key writes in, 18
 - full-text indexing, 106, 244
 - I/O, tuning, 281–283
 - indexes in, 18, 113–117
 - building quickly, 148
 - caching, 149
 - writes to, deferring, 281
 - key caches (key buffers), 269, 274–277, 561
 - locking in, 14, 18
 - memory mapping, 283
 - packed indexes, 126
 - recovering from corruption, 282
 - redundant indexes, 128
 - repair of tables in, 18
 - restoring raw files, 500
 - spatial indexes, 106
 - table locks, 149
 - transactions not supported, 149
- mysam_block_size variable, 277
- mysam_recover variable, 282
- mysam_use_mmap variable, 283
- mylvmbackup tool, 492, 515, 517

MySQL

- alternatives to, 471
- architecture of, 1–3
- benchmarking (see single-component benchmarking)
- in chrooted environment, 554
- connecting to, troubleshooting, 76–79, 533
- developers of, contacting, 470
- documentation for, 559
- extending, 470
- privileges for running, 541
- profiling, 63–70
- source code for, 554
- upgrades for
 - changes to optimizer in, 203
 - testing with replication, 345
- MySQL Administrator, 584
- MySQL Benchmark Suite (sql-bench), 43, 53
- mysql database (see database)
- MySQL Forge community site, 582, 601
- MySQL Master-Master Replication Manager tool, 454
- MySQL Migration Toolkit, 584
- MySQL Monitoring and Advisory Service, 588
- MySQL Proxy, 76, 443, 470, 598
- MySQL Query Browser, 583
- MySQL server (see server)
- MySQL Statement Log Analyzer (mysqsla), 69, 596
- MySQL Stored Procedure Programming (Harrison, Feuerstein), 217
- MySQL visual tools, 583
- MySQL Workbench, 584
- MySQL, version 4.1, password hashing scheme, 529
- MySQL, version 5.0
 - patch, for removing verbose record dumps, 657
 - privileges, changes in, 529–532
 - SHOW STATUS command, 558
 - stored routines, 530
 - triggers, 530
- MySQL, version 5.1
 - full-text searching, changes in, 248
 - INFORMATION_SCHEMA database, 558
 - patch for slow query times, 66
- mysql_slow_log_filter tool, 69
- mysql_slow_log_parser tool, 69

- mysqladmin utility
 - debug command, 653
 - drop command, 540
 - extended command, 70, 276, 558, 559, 564
 - flush-hosts command, 549
 - shutdown command, 519
 - variables command, 557
- mysql-bin.index file, 357, 488
- mysqlbinlog tool, 487
- mysqldump tool
 - initializing slave using, 353
 - logical backups using, 489, 511, 517
 - resyncing slave using, 381
 - table conversions using, 30
- mysqldumpslow tool, 69
- mysqlhotcopy tool, 352, 513, 517
- mysqlmanager tool, 266
- mysqldump tool, 492
- mysql-relay-bin.index file, 357
- mysqlreport tool, 301, 595
- mysqsla (MySQL Statement Log Analyzer), 69, 596
- mysqslap tool, 43
- mysqsniffer tool, 76
- mytop tool, 591

N

- Nagios System and Network Monitoring (Barth), 587
- Nagios tool, 587
- name locks, 651, 654
- NAS (network-attached storage), 326
- Native POSIX Threads Library (NPTL), 334
- natural-language full-text searches, 245–247
- nc tool, 605, 606
- NDB API, 471
- NDB Cluster storage engine, 22, 29, 435
 - configuration, status variables for, 564
 - locking in, 14
 - T-Trees used for indexes, 97
- NDB module for Apache, 471
- Ndb_* status variables, 564
- netstat tool, 77, 336
- network
 - access to, disabling, 543
 - configuration of, 328–330
 - latency of, 458
 - monitoring, 329
 - security for, 542–550

- network-attached storage (NAS), 326
- next-key locking strategy, InnoDB, 19
- Nichter, Daniel (HackMySQL tools web site), 595
- nondeterministic functions, caching not used for, 205
- noninteractive mode, 591
- noninteractive monitoring tools, 586–590
- nonrepeatable read, 8
- normalization, 139–140, 141
- Not_flushed_delayed_rows status variable, 564
- NOW() function, caching not used for, 205
- NOW_USEC() UDF, 230
- NPTL (Native POSIX Threads Library), 334
- NTFS filesystem, 333
- nullable data types, 81

O

- O_DIRECT flag, 289
- O_DSYNC flag, 290
- O_SYNC flag, 289
- object hierarchies, for caching, 468
- object versioning, 468
- Object-relational mapping (ORM), 96
- object-specific privileges, 522
- obsolete privileges, 540
- offline backups, 478
- OFFSET clause, optimizing, 193
- OLAP (online analytical processing),
 - separating from OLTP, 372
- OLD_PASSWORD() function, 529
- old_passwords variable, 529
- OLTP (online transaction processing)
 - performance of, measuring, 35
 - separating from OLAP on different slaves, 372
 - sysbench tool for, 49–50
- on-controller cache (RAID cache), 323
- on-disk caches, 466
- online analytical processing (OLAP), 372
- online backups, 478
- online transaction processing (see OLTP)
- Open_* status variables, 561
- Open_files status variable, 303
- Open_tables status variable, 303
- Opened_tables status variable, 280, 303
- OpenNMS tool, 588
- OpenSSL library, 546
- operating system
 - backing up files on, 482
 - choosing, 330
 - memory requirements for, 273
 - profiling, 76–79
 - security of, 541
 - status of, monitoring, 336–341
 - for CPU-bound server, 339
 - for idle server, 341
 - for I/O-bound server, 340
 - iostat tool, 338–339
 - for swapping server, 341
 - vmstat tool, 336
 - updating, importance of, 541
- operating system waits, 566
- operations accounts, 528
- OProfile tool, 78
- optimal concurrency, 462
- optimization
 - BLOB columns, 298–301
 - cache tables, 142–145
 - counter tables, 144
 - of CPUs, 306–309
 - data types, 80
 - bit-packed, 91–93
 - choosing, 81
 - date and time, 82, 90
 - for identifier columns, 93
 - nullable, 81
 - real numbers, 83
 - size of, 81, 86
 - strings, 84–90
 - whole numbers, 82
 - denormalization, 139, 140–142
 - of filesorts, 176, 300
 - filesystem for, choosing, 331–333
 - fragmentation, reducing, 138
 - of full-text searching, 251–252
 - index corruption, repairing, 136
 - index statistics, updating, 136
 - of memory-to-disk ratio, 314–317
 - of multiple disk volumes, 327
 - of network configuration, 328–330
 - normalization, 139–140, 141
 - operating system for, choosing, 330
 - of RAID, 317–325
 - of slave hardware, 317
 - of sorts, 135, 176
 - summary tables, 142–145
 - table corruption, repairing, 136
 - TEXT columns, 298–301
 - (see also availability, high; load balancing; performance; scaling; server configuration)

OPTIMIZE TABLE command, 138
 optimizer_prune_level variable, 198
 optimizer_search_depth variable, 197
 order processing, storage engines suited for, 27
 ORM (object-relational mapping), 96

P

packed (prefix-compressed) indexes, 126
 packet loss, 328
 packet sniffers, for profiling, 76
 parallel dump and restore, 491
 parallel execution, not supported, 185
 parallel result sets, Sphinx tool for, 631
 partitioned data (see data sharding)
 partitioned tables, 253, 257–262
 advantages of, 258
 examples of, 259
 limitations of, 260
 queries against, optimizing, 261
 for scalability, 434
 types of, 258
 partitioning keys, for data sharding, 419–421
 partitioning, functional, 415, 416, 446
 passive caches, 463
 passive monitoring, 585
 passwordless access, disallowing, 534
 passwords
 hashing, 529, 550
 security of, 527
 pattern matching, limitations of, 536
 PBXT (Primebase XT) storage engine, 23, 29
 locking in, 14
 MVCC supported by, 12
 percent sign (%), default hostname, 526, 534
 percentile response times, 35
 performance, 306, 410, 413
 of ALTER TABLE command, 145–148
 of application
 caching for, 463–469
 extending MySQL for, 470
 optimal concurrency, finding, 462
 problems with, finding, 457–460
 web server problems, 460–463
 application level profiling affecting, 57
 autogenerated schemas affecting, 96
 of backup and recovery, 510
 of character sets and collations, 241–244
 of cursors, 224
 development complexity increased by
 improving, 145
 of distributed (XA) transactions, 263, 264
 of DNS, avoiding reliance on, 329
 of EXPLAIN command, 608
 of file copies, 606
 of foreign keys, 252
 of full-text searching, 249–252
 of indexes
 building quickly, 148
 clustered indexes, 110–119
 covering indexes, 120–124
 duplicate indexes, 127–129
 index scans for sorts, 124–126
 isolating columns, 106
 locking and, 129–131
 packed indexes, 126
 prefix indexes, 107–110
 redundant indexes, 127–129
 sorting, 135
 of locks, 5
 of merge tables, 255
 of OLTP, measuring, 35
 of partitioned tables, 261
 of prepared statements, 226
 privileges affecting, 532
 of queries
 data access, analyzing, 152–157
 optimizations for specific types
 of, 188–195
 optimizer for, 3, 165–188, 195, 203
 query cache for, 164
 restructuring queries for, 157–160
 (see also query cache)
 of replication, 405–407
 server configuration benefits for, 265
 of server variables set
 dynamically, 268–270
 in Sphinx, controlling, 641
 of stored code, 217–223
 swapping affecting, 334
 of temporary tables, 298
 tools for
 analysis tools, 595–598
 Dormando's Proxy for MySQL
 tool, 599
 interface tools, 583–585
 Maatkit tools, 599
 monitoring tools, 585–595
 MySQL Proxy, 598

- performance (*continued*)
 - of UDFs, 230
 - of views, 232, 234
 - (see also benchmarking; optimization; profiling)
- performance counters, 574
- permissions, 522
 - (see also privileges)
- perrow utility, 280
- persistent connections, compared to
 - connection pooling, 460
- phantom read, 8
- phpMyAdmin tool, 543, 585
- phrase proximity ranking, in Sphinx, 637
- phrase searches, 247
- physical reads, 311
- physical size, of hard disk, 316
- Planet MySQL blog aggregator, 601
- plug-in-specific status variables, 564
- point-in-time recovery, 504
- Practical Unix and Internet Security*
 - (Garfinkel et al.), 541
- prefix indexes, 107–110
- prefix-compressed (packed) indexes, 126
- preforking, 460
- prepared statements, 225
 - limitations of, 229
 - optimizing, 227
 - SQL interface for, 227
- Preston, Curtis (*Backup & Recovery*), 472
- primary key order, inserting records
 - in, 117–120
- Primebase XT storage engine (see PBXT storage engine)
- privileges, 522
 - adding, 525, 526–529
 - for adding privileges, 537
 - amount of, 532
 - for backup accounts, 528
 - for database administrator accounts, 527
 - displaying for user, 525
 - for employee accounts, 527
 - for INFORMATION_SCHEMA tables, 532
 - global, 522
 - how MySQL checks, 524
 - invisible, 537–540
 - for logging accounts, 528
 - for monitoring accounts, 528
 - for mysql database, 535
 - object-specific, 522
 - obsolete, 540
 - for operations accounts, 528
 - performance affected by, 532
 - removing, 525, 526, 537
 - for removing privileges, 537
 - for replication accounts, 348
 - for running MySQL, 541
 - stored in grant tables, 523
 - for stored routines, 530
 - for system administrator account, 527
 - for temporary tables, 534
 - for triggers, 530
 - troubleshooting, 533–541
 - types of, 522
 - for views, 531
 - for wildcarded databases, 536
 - (see also authorization; permissions)
- /proc filesystem, 78
- procedures, stored, 219, 228
- processes, troubleshooting, 76–79
- procs_priv table, 524
- production environment, isolating, 542
- profiling, 32, 54
 - application level, 55–63, 457
 - example of, 56–63
 - lightweight, 57
 - measurements for, 55
 - performance affected by, 57
 - CPU usage, tools for, 63
 - MySQL, 63–70
 - MySQL server, 70–75
 - operating system, 76–79
 - packet sniffers for, 76
 - proxies for, 76
 - queries, 71–74
 - web server logs for, 76
 - without code or patches, 76
 - (see also SHOW INNODB STATUS command; SHOW MUTEX STATUS command)
- proxies, for profiling, 76
- PURGE MASTER LOGS command, 378
- purging data, for scalability, 432–435
- Putty tool, 548
- pyramid (tree) topology, replication, 370

Q

- Qcache_* status variables, 303, 561
- Qcache_hits status variable, 209
- Qcache_inserts status variable, 210
- Qcache_not_cached status variable, 210

queries

- accelerating (see Sphinx tool)
- access methods used by, 155
- analyzing, 152–157, 596, 598
- character sets affecting, 241–244
- client/server protocol for, 161–164
- collations affecting, 241–244
- consistent formatting of, importance of, 205
- COUNT() queries, 167, 188–190
- DISTINCT clauses, 191
- execution of, 160, 178
- execution plan for, 172, 178
 - (see also EXPLAIN command)
- execution time of, 154
- GROUP BY clause, 191, 630, 646
- IN() list comparisons, 169
- index-covered queries, 121
- joins
 - decomposition of, 159
 - execution strategy for, 170–172
 - optimizations for, 167, 173–176, 190
 - STRAIGHT_JOIN option, 196
- LIMIT clause, 193, 194
- MAX() queries, 167, 187
- MIN() queries, 167, 187
- monitoring, 591
- OFFSET clause, 193
- optimizations for specific types of, 188–195
- optimizer for, 3, 165–170
 - dynamic optimizations, 166
 - early termination by, 168
 - hint options for, 194, 195, 203
 - index statistics used by, 170
 - join optimizations by, 173–176
 - join strategy used by, 170–172
 - limitations of, 165, 179–188
 - optimizations performed by, 167–169
 - sort optimizations used by, 176
 - static optimizations, 166
 - system variables affecting, 197
 - table statistics used by, 170
- parser for, 165
- of partitioned tables, 261
- prepared statements for, 225
 - limitations of, 229
 - optimizing, 227
 - SQL interface for, 227
- preprocessor for, 165
- profiling, 71–74

- restructuring, 157–160
 - breaking into multiple queries, 157
 - join decomposition, 159
 - reducing rows returned by, 158
- results returned by, 178
- rows examined by, 154–157
- rows returned by, 153, 154, 158
- software for, 583
- states of, 163
- subqueries, 168, 191
- subqueries, correlated, 179–183
- UNION clause, 183, 194, 254
- WHERE clause, 169, 628
- query cache, 3, 164, 204–209
 - cache hits
 - checking for, 205
 - improving, 213, 216
 - cache misses, reasons for, 209
 - column privileges not using, 532
 - disabling, 211, 216
 - enabling, 211
 - excluding queries from, 216
 - fragmentation in, 208, 212
 - helpfulness of, determining, 209–211
 - hit rate of, 209
 - locks affecting, 212
 - memory use by, 206–209, 211
 - not using for query, 71
 - overhead added by, 206
 - pruning of, 213, 216
 - removing all queries and results from, 213
 - result sets in, size of, 212
 - size of, 269
 - allocated, 211
 - performance affected by, 206
 - potential, 211
 - status variables for, 561
 - tuning, 211–213
- query logs, 64–70
- query plan cost, status variable for, 564
- Query state, 163
- query_cache_limit variable, 212
- query_cache_min_res_unit variable, 207, 210, 211, 212
- query_cache_size variable, 211, 216, 267, 269
- query_cache_type variable, 211
- query_cache_wlock_invalidate variable, 212
- query-based split, 439

question mark (?), parameters in prepared statements, 225
 Questions status variable, 560
 quotes ('...'), for hostnames and usernames, 535

R

R1Soft, 517

RAID (Redundant Arrays of Inexpensive Disks), 317–325

as backups, 475

BBU (battery backup unit), 324, 327

configuration, 322–325

crash testing script for, 324

failure of, 320

hardware, balancing with software, 321

levels of, 318–320

monitoring, 320

stripe chunk size for, 322

RAID cache, 323

random I/O, 310

random, load-balancing algorithm, 444

range condition, 134

range partitioning, 258

ranged queries, in Sphinx, 641

raw backups, 476, 480, 500

READ COMMITTED isolation level, 8, 14

read locks, 4

READ UNCOMMITTED isolation level, 8, 14

read_buffer_size variable, 269, 270

read_only variable, 354

read_rnd_buffer_size variable, 270

read-around writes, 276

reading (see I/O)

read-mostly tables, storage engines suited for, 27

read-only slaves, 373

read-only tables, storage engines suited for, 27

real numbers, data types for, 83

records_in_range() function, 136

recovery, 473–475, 499–510

delayed replication for, 506

disaster recovery, 476

with InnoDB, 507–510

limiting MySQL access during, 500

log server for, 506

logical backups, restoring, 502–504

point-in-time recovery, 504

raw files, restoring, 500

speed of, 510

redundancy, adding, for high availability, 449

Redundant Arrays of Inexpensive Disks (see RAID)

redundant indexes, 127–129, 597

ReiserFS filesystem, 332, 333

relay log, 346

relay_log variable, 349

relay_log_space_limit variable, 355

relay-log.info file, 358

RELOAD privilege, 528

REPAIR TABLE command, 136

REPEATABLE READ isolation level, 8, 14

replicate_* variables, 360

replicate_ignore_db variable, 372

replicated-disk architectures, 449

replication, 343–346

accounts for, creating, 347

for backups, 345, 475, 485

capacity planning for, 376–378

configuration for, 348

backing up, 482

recommended, 353

CPUs affecting, 307

delayed, for recovery, 506

files used by, 357

filtering for, 360

for scalability, 415

future of, 407

limitations of, 405

for load balancing, 438

monitoring, 378, 591, 598

performance of, 405–407

row-based, 343, 356

Sandbox script for experimentation with, 352

scaling reads using, 344

scaling writes using, 344, 377

setting up, 347–355, 360

slave lag, measuring, 379

slave server

changing master of, 382

consistency with master, determining, 380

delaying (lagging), 600

hardware for, 317

initializing from another server, 352

as master of other slaves, 359

prefetch for, 600

- promoting to master, 382–387
- restarting, 600
- resyncing from master, 381
- starting, 349–352
- speed of, measuring, 230
- statement-based, 343, 355
- status of, 580
- stored routines and, 219
- synchronous, for high availability, 451
- topologies for, 362
 - customizing, 371
 - data archiving, 372
 - full-text searches, 373
 - log server, 374–376
 - master and multiple slaves, 362
 - master, distribution master, and slaves, 369–370
 - master-master with slaves, 367
 - master-master, in active-active mode, 363–364
 - master-master, in active-passive mode, 365, 387
 - multimaster, 364, 373
 - read-only slaves, 373
 - ring, 367
 - selective replication, 371
 - separating functions, 372
 - tree (pyramid), 370
- troubleshooting
 - all updates not replicated, 395
 - bandwidth, limited, 404
 - data changes on slave, 392
 - data corruption or loss, 388–390
 - dependencies on nonreplicated data, 394
 - disk space, running out of, 404
 - InnoDB locking SELECTs, 396–398
 - locking contentions, 396–398
 - mixing transactional and nontransactional tables, 391
 - nondeterministic statements, 392
 - nontransactional table errors, 391
 - nonunique server IDs, 393
 - packets from master, oversized, 404
 - setup problems, 360
 - slave lag, excessive, 399–404
 - storage engines different on master and slave, 392
 - temporary tables, missing, 394
 - undefined server IDs, 393

- writing to both masters, 398
- underutilization of servers, planning, 377
- uses of, 344
- in versions prior to 4.0, 346
- versions of MySQL used with, mixing, 344
- REQUIRE ISSUER option, 547
- REQUIRE SUBJECT option, 547
- RESET QUERY CACHE command, 213
- resources (see books and publications; web site resources)
- response time (see latency)
- restoring data, 473
- reuse of code, 217
- REVOKE command, 525, 526, 537
 - for global privileges, 537
 - not replicating, 361
- Richter, Georg (patch for slow query times), 66
- ROLLBACK command, 6
- round-robin, load-balancing algorithm, 444
- row fragmentation, 138
- row locks, 6, 14, 150
- ROW OPERATIONS section, SHOW INNODB STATUS output, 577
- row operations, status of, 577
- row-based replication, 343, 356
- RRDTool-based systems, 589
- rsync tool, 605, 606
- R-tree (spatial) indexes, 106

S

- SAN (storage area network), 325
- Sandbox script, 352
- sar tool, 336
- scalability, 409, 410, 412
 - active data, separating from inactive data, 434
 - clustering for, 435
 - intermediate remedies before scaling, 413
 - load balancing for, 436–447
 - planning for, 412
 - scaling back, 432–435
 - scaling horizontally (out), 412, 415
 - data sharding for, 417–432
 - partitioning for, 415, 416
 - replication for, 415
 - scaling vertically (up), 412, 414
- scalability measurements, 35
- scanning indexes, 124–126

- schema (see database)
- Schwartz, Baron
 - innotop tool, 591
 - Maatkit tools, 596
- scp tool, 604, 606
- scripting backups, 518–520
- searchd program, in Sphinx, 635
- searching, full-text (see full-text searching)
- Seconds_Behind_Master status
 - variable, 351, 379
- Secure Sockets Layer (see SSL)
- secure_auth variable, 529
- security
 - application-level encryption, 552–554
 - automatic host blocking, 549
 - of backups, 476
 - chrooted environment, MySQL in, 554
 - connection encryption, 545–548
 - data encryption, 550–554
 - DMZs, 545
 - filesystem encryption, 551
 - firewalls, 544
 - hashing passwords, 529, 550
 - localhost-only connections, 543
 - of network, 542–550
 - of operating system, 541
 - of passwords, 527
 - source code modification for, 554
 - SSL, 546, 563
 - TCP wrappers, 548
 - tunneling, 545, 548
 - (see also access control; authentication; authorization; privileges)
- SELECT command
 - status variables for, 561
 - with UPDATE, not supported, 187
 - (see also queries)
- SELECT INTO OUTFILE command, 240, 490, 491
- SELECT privilege, 522, 535
- Select_full_join status variable, 303, 562
- Select_full_range_join status variable, 303, 562
- Select_range status variable, 561
- Select_range_check status variable, 303, 562
- Select_scan status variable, 558, 561
- selective replication, 371
- selectivity of indexes, 107–110
- SEMAPHORES section, SHOW INNODB
 - STATUS output, 566
- Sending data state, of query, 164
- sequential I/O, 310
- seqwr benchmark, sysbench tool, 50
- SERIALIZABLE isolation level, 9, 14
- server
 - auditing, 542
 - grouping servers, 591
 - lock waits in, 650–655
 - profiling, 70–75
 - restricting logins on, 541
- server administration, software for, 584, 585
- server configuration
 - backing up, 482
 - benchmarking prior to, 270
 - BLOB columns, 298–301
 - buffer pool, 271
 - changing dynamically, 267, 268–270
 - changing gradually, 270
 - concurrency tuning, 295–297
 - data dictionary, 280
 - default settings for, 265
 - files for, 266, 267
 - filesorts, 300
 - key caches, 274–277
 - for memory usage, 272–274
 - per-connection settings, 304
 - performance benefits from, 265
 - for replication, 348, 353
 - sample files for, 271
 - scope of settings in, 267
 - status variables, 301–304
 - syntax used in, 267
 - table cache, 279
 - TEXT columns, 298–301
 - thread cache, 278
 - units used in, 268
 - workload-based tuning, 298–304
 - (see also I/O)
- server status (see status of MySQL server)
- server variables, viewing, 591
- /server-status/ URL, 78
- session-based split, 439
- SET CHARACTER SET command, 238
- SET command, 267
- SET NAMES command, 238
- SET TRANSACTION ISOLATION LEVEL
 - command, 11
- SET type, 92, 94
- SHA1() function, 550
- sharding (see data sharding)
- shared hosting provider, backups by, 477
- shared locks (read locks), 4

- shared-storage architectures, 449
- SHOW BINARY LOGS command, 581
- SHOW BINLOG EVENTS command, 378, 581
- SHOW CHARACTER SET command, 240
- SHOW COLLATION command, 240
- SHOW CREATE TABLE command, 535
- SHOW DATABASES privilege, 536, 538
- SHOW ENGINE INNODB STATUS
 - command, 565
- SHOW FULL PROCESSLIST
 - command, 163
- SHOW GLOBAL STATUS command, 301, 558
- SHOW GLOBAL VARIABLES
 - command, 268
- SHOW GRANTS command, 525, 537–540
- SHOW INNODB STATUS
 - command, 565–578, 656
 - BUFFER POOL AND MEMORY
 - section, 576
 - FILE I/O section, 574
 - INSERT BUFFER AND ADAPTIVE
 - HASH INDEX section, 575
 - LATEST DETECTED DEADLOCK
 - section, 569–571
 - LATEST FOREIGN KEY ERROR
 - section, 567–569
 - LOG section, 576
 - ROW OPERATIONS section, 577
 - SEMAPHORES section, 566
 - TRANSACTIONS section, 572–574, 656
- SHOW MASTER LOGS command, 375
- SHOW MASTER STATUS command, 348, 378, 580
- SHOW MUTEX STATUS command, 579
- SHOW PROCESSLIST command, 71, 76, 578, 650
- SHOW PROFILE patch, 74
- SHOW SESSION STATUS command, 71
- SHOW SLAVE STATUS command, 350, 379
- SHOW STATUS command, 70, 558–565, 595
- SHOW TABLE STATUS command, 14
- SHOW USER STATISTICS command, 400
- SHOW VARIABLES command, 557
- shutdown command, mysqldadmin, 519
- SHUTDOWN privilege, 521
- single-component benchmarking, 33, 34, 43
- single-pass sort algorithm, 177
- skip_grant_tables variable, 536
- skip_name_resolve variable, 329, 532
- skip_networking variable, 543
- skip_slave_start variable, 354
- Slave_* status variables, 564
- slave_compressed_protocol variable, 404
- slaves (see replication, slave server)
- Sleep state, of query, 163
- SLEEP() function, 652
- slow query log, 64–68
- Slow_launch_threads status variable, 303
- Slow_queries status variable, 562
- slow_query_log variable, 64
- slow_query_log_file variable, 64
- SMALLBLOB type, 86
- SMALLINT type, 82
- SMALLTEXT type, 86
- Smokeping tool, 329
- SNAP Innovation GmbH, 23
- snapshot-based backups, 476
- snapshots, filesystem, 492–499
 - (see also LVM)
- software RAID, 321
- software, finding, resources for, 601
- Solaris operating system, 330
- Solid Information Technology, 23
- solidDB storage engine, 14, 23, 29
- sort_buffer_size variable, 267, 270, 304, 562
- Sort_merge_passes status variable, 300, 303, 562
- Sort_range status variable, 563
- Sort_scan status variable, 563
- sorting
 - filesorts, 176, 300
 - index scans for, 124–126
 - optimizing, 135, 176
 - status variables for, 562
- Sorting result state, of query, 164
- Souders, Steve (*High Performance Web Sites*), 461
- spatial (R-tree) indexes, 106
- Sphinx tool, 623, 637
 - attributes support, 638
 - for data sharding, 432
 - examples using, 623–626, 643–648
 - filtering in, 639
 - full-text searching with, 627, 643–645
 - GROUP BY queries, optimizing, 630, 646
 - indexer program in, 635
 - installing, 635
 - MVAs supported by, 639

Sphinx tool (*continued*)

- parallel result sets using, 631
- partitioning in, 636
- performance control with, 641
- phrase proximity ranking, 637
- ranged queries in, 641
- reasons to use, 627
- scalability of, 632
- searchd program in, 635
- sharded data, aggregating, 634, 648
- top results in order, finding, 629
- WHERE clause, improving efficiency of, 628

- SphinxSE storage engine, 640–641

- spindle rotation speed, of hard disk, 316

- SQL dumps, 489

- SQL SECURITY DEFINER

- characteristic, 530

- SQL SECURITY INVOKER

- characteristic, 530

- SQL slave thread, 346

- SQL_BIG_RESULT option, 196

- SQL_BUFFER_RESULT option, 196

- SQL_CACHE option, 196, 216

- SQL_CALC_FOUND_ROWS option, 194, 197

- SQL_NO_CACHE option, 71, 196, 216

- SQL_SMALL_RESULT option, 196

- sql-bench (MySQL Benchmark Suite), 43, 53

- SQLyog tool, 584

- ssh tool, 606

- SSH tunneling, 548

- SSL (Secure Sockets Layer), 546, 563

- Ssl_* status variables, 563

- stale-data split, 439

- Starkey, Jim (developer of Falcon), 23

- START SLAVE command, 350

- START TRANSACTION command, 6

- statement handle, 225

- statement-based replication, 343, 355

- statements, prepared (see prepared statements)

- static optimizations, for queries, 166

- Statistics state, of query, 164

- status of MySQL server

- binary logs, 581

- connections, list of, 578

- determining, methods for, 557

- INFORMATION_SCHEMA views for, 581

- InnoDB status, 565–578

- adaptive hash index, 575

- buffer pool, 576

- current waits, 566

- deadlocks, 569–571

- event counters, 566

- foreign key errors, 567–569

- helper threads, 574

- insert buffer, 575

- mutexes, 579

- performance counters, 574

- row operations, 577

- transaction logs, 576

- transactions, 572–574

- replication, 580

- status variables, 301–304, 558

- binary logging, 559

- command counters, 560

- connections, 559

- distributed (XA) transactions, 564

- file descriptors, 561

- handler operations, 560

- InnoDB, 563

- INSERT DELAYED queries, 564

- MyISAM key buffer, 561

- NDB Cluster configuration, 564

- plug-in-specific, 564

- query cache, 561

- query plan cost, 564

- SELECT queries, 561

- sorting, 562

- SSL, 563

- table locking, 563

- temporary files and tables, 560

- threads, 559

- system variables for, 557

- status of operating system,

- monitoring, 336–341

- stock quotes, storage engines suited for, 28

- stopwords, 244, 251

- storage area network (SAN), 325

- storage capacity, of hard disk, 316

- storage engine API, 2

- storage engines, 2, 10, 29

- choosing for an application, 24–29

- consistency of backups with, 483–485

- converting tables between, 30

- creating, 470

- determining for a table, 14

- list of, including features, 29

- lock waits in, 655–658

- mixing in transactions, 11

- third-party, 24

- (see also specific storage engines)

storage_engine variable, 557
stored code
 advantages of, 217
 comments in, 224
 disadvantages of, 218
 events, 222
 language constructs used in, 217
 library of, 217
 stored functions, 219
 stored procedures, 219, 228
 triggers, 220–222
stored functions, 219
stored procedures, 219, 228
stored routines, privileges used with, 530
strace tool, 76, 78
STRAIGHT_JOIN option, 196
string locks, 651
strings
 data types for, 84–90
 for identifier columns, 94
Stunnel tool, 548
subqueries
 correlated, optimization of, 179–183
 optimizations for, 168, 191
summary tables, 142–145
SUPER privilege
 for operations and monitoring
 accounts, 529
 and read_only option, 354
 for triggers, 531
 when to grant, 536
Super Smack, 44
surrogate keys, 117
swapping, 334, 341
sync_binlog variable, 294, 328, 354
synchronization of data (see replication)
synchronous replication, for high
 availability, 451
sysbench tool, 43, 46–50
system administrator account, 527
system security, 541
system variables
 affecting query optimizer, 197
 exposing, 557

T

table cache, 269, 279
table conversions, between storage
 engines, 30
table definition cache (see data dictionary)
table locks, 5, 149, 650, 651–653

 concurrency level of, 14
 as potential bottleneck, 149
 status variables for, 563
table statistics, 170
table_cache variable, 268
table_cache_size variable, 269
table_definition_cache variable, 280
Table_locks_immediate status variable, 563
Table_locks_waited status variable, 304, 563
table_open_cache variable, 280
tables
 checksums of, 600
 corruption of, 136
 filename of, 14
 information about, displaying, 14
 storage engine used by, determining, 14
 synchronizing, 600
tables_priv table, 524
tablespace, InnoDB, 19, 290–293
tagged cache, 468
tar command, 605
Tc_log_* status variables, 564
TCP wrappers, 548
TCP/IP Network Administration (Hunt), 542
tcpdump tool, 76
temporary files, status variables for, 560
temporary tables
 avoiding, 87
 compared to Memory tables, 21
 improving performance of, 298
 privileges for, 534
 status variables for, 560
TEMPTABLE algorithm, for views, 232
TEXT types, 86, 298–301
thread cache, 269, 278
thread libraries, 334
thread_cache_size variable, 269, 278
threaded discussion forums, storage engines
 suited for, 28
threads (see connections to MySQL)
threads benchmark, sysbench tool, 50
Threads_connected status variable, 279, 558
Threads_created status variable, 278, 304,
 559
throughput, 34, 307, 315
time data types (see date and time data types)
time to live (TTL), cache control policy, 467
time-based data partitioning, 434
TIMESTAMP type, 90
 compared to DATETIME type, 82
 high-resolution support, 91

- TINYBLOB type, 86
- TINYINT type, 82
- TINYTEXT type, 86
- TPC-C test, 34, 51
- transaction logs, 10, 284–285, 576
- transactions, 6–12
 - ACID test for, 7
 - AUTOCOMMIT mode for, 11
 - choosing storage engine based on, 25
 - DDL commands committing
 - automatically in, 11
 - deadlocks of, 9, 569–571, 598
 - features listed by storage engine, 29
 - isolation levels for, 8, 11, 14, 19
 - mixing storage engines in, 11
 - monitoring, 591
 - query cache affected by, 206, 215
 - status of, 572–574
- transactions per time unit (throughput), 34
- TRANSACTIONS section, SHOW INNODB
 - STATUS output, 572–574
- transfer speed, of hard disk, 315, 316
- tree (pyramid) topology, replication, 370
- TRIGGER privilege, 531
- triggers, 220–222
 - privileges used with, 530
 - (see also stored code)
- troubleshooting
 - application performance
 - caching, 463–469
 - problems with, finding, 457–460
 - web server problems, 460–463
 - connection errors, 533
 - connections, 76–79
 - data fragmentation, 138
 - index corruption, 136
 - index fragmentation, 138
 - locks, 650–658
 - MySQL upgrades, problems introduced
 - by, 203
 - privileges, 533–541
 - processes, 76–79
 - replication
 - all updates not replicated, 395
 - bandwidth, limited, 404
 - data changes on slave, 392
 - data corruption or loss, 388–390
 - dependencies on nonreplicated
 - data, 394
 - disk space, running out of, 404
 - InnoDB locking SELECTs, 396–398

- locking contentions, 396–398
- mixing transactional and
 - nontransactional tables, 391
- nondeterministic statements, 392
- nontransaction table errors, 391
- nonunique server IDs, 393
- packets from master, oversized, 404
- setup problems, 360
- slave lag, excessive, 399–404
- storage engines different on master and
 - slave, 392
- temporary tables, missing, 394
- undefined server IDs, 393
- writing to both masters, 398
- table corruption, 136
- TTL (time to live), cache control policy, 467
- T-Tree indexes, 97
- tunneling, 545, 548
- two-pass sort algorithm, 177

U

- UDFs (see user-defined functions)
- UFS filesystem, 333
- UFS2 filesystem, 333
- unarchiving, 433
- UNION clause, 183, 194, 254
- UNLOCK TABLES command, 12, 652
- UNSIGNED attribute, 82
- updatable views, 233
- UPDATE command
 - EXPLAIN command with, 609
 - with SELECT, not supported, 187
- upgrades
 - changes to optimizer in, 203
 - testing with replication, 345
- Uptime status variable, 564
- USAGE privilege, 537
- USE INDEX option, 197
- user locks, 655
- user table, 523
- user-defined functions (UDFs), 230, 470
- user-defined variables, 198–203
- usernames
 - quoting in commands, 535
 - uniqueness of, 522, 535
- users (see accounts)
- UUID values
 - generating, 399
 - inserting, 117–118
 - storing, 94, 95

V

VARCHAR type, 84–86, 89
 variables command, mysqladmin, 557
 variables, server (see server variables)
 variables, status (see status variables)
 variables, system (see system variables)
 variables, user-defined, 198–203
 version-based split, 439
 views, 231

- limitations of, 236
- materialized, 236
- MERGE algorithm for, 232
- performance of, 232, 234
- privileges used with, 531
- TEMPTABLE algorithm for, 232
- updatable, 233

 virtual private network (VPN), 546
 vmstat tool, 335, 336
 VPN (virtual private network), 546

W

Wackamole, 438
 waits, lock (see lock waits)
 waits, operating system, 566
 “warm” backups, 473
 web server logs, for profiling, 76
 web server problems, 460–463
 web site resources

- ab tool, 42
- Cacti tool, 329, 590
- crash testing script, 324
- Cricket, 590
- Database Test Suite, 43
- Dormando’s Proxy for MySQL, 599
- DRBD, 449
- ESI (edge side includes), 461
- functions for memcached, 230
- Groundwork Open Source, 588
- HackMySQL tools, 596
- Hibernate Shards, 432
- High Availability Linux project, 452
- HiveDB, 432
- http_load tool, 42
- Hyperic HQ, 588
- InnoDB Recovery Toolkit, 510
- innotop, 591, 595
- JMeter tool, 42
- library of stored routines, 217
- LVS (Linux Virtual Server), 438

Maatkit tools, 596, 600
 MONyog tool, 589
 MRTG (Multi Router Traffic
 Grapher), 329, 589
 mtop, 591
 Munin, 590
 mylvmbackup tool, 515
 MySQL Benchmark Suite (sql-bench), 44
 MySQL developers, 470
 MySQL documentation, 559
 MySQL Forge community site, 582, 601
 MySQL Master-Master Replication
 Manager tool, 454
 MySQL Monitoring and Advisory
 Service, 589
 MySQL Proxy, 598
 MySQL Statement Log Analyzer
 (mysqlsla), 69
 MySQL visual tools, 584
 mysql_slow_log_filter tool, 69
 mysql_slow_log_parser tool, 69
 mysqldump tool, 492
 mysqlslap tool, 43
 mysqlsniffer tool, 76
 mytop, 591
 Nagios, 587
 NDB API, 471
 NDB module for Apache, 471
 OpenNMS, 588
 OProfile tool, 78
 packet sniffers, 76
 patch for removing verbose record
 dumps, 657
 patch for slow query times, 66
 phpMyAdmin, 585
 Planet MySQL blog aggregator, 601
 Putty tool, 548
 R1Soft, 517
 RRDTool, 589
 Smokeping tool, 329
 SNAP Innovation GmbH, 23
 Solid Information Technology, 23
 Sphinx, 623, 649
 SQLyog, 585
 SSH tunnels, connecting to MySQL,
 tutorial for, 548
 Super Smack, 44
 sysbench tool, 43
 tcpdump tool, 76
 TPC-C test, 34

- web site resources (*continued*)
 - user-defined function examples, 230
 - Wackamole, 438
 - Zabbix, 588
 - Zenoss, 588
 - ZRM (Zmanda Recovery Manager), 515–516
- weighted, load-balancing algorithm, 444
- WHERE clause
 - propagation of, 169
 - Sphinx improving efficiency of, 628
- whole numbers, date types for, 82
- Widenius, Michael (developer of Maria engine), 24
- wildcarded databases, privileges for, 536
- Windows operating system, 331
- WITH ROLLUP clause, optimizations using, 193
- with-libwrap variable, 549
- working concurrency, 37
- working set of data, 312
- workload partitioning (see functional partitioning)
- workload-based tuning, 298–304

- write capacity, increasing, 419
- write locks, 4
- write-ahead logging, 312
- writing (see I/O)

X

- XA transactions (see distributed (XA) transactions)

- XFS filesystem, 332, 333

Y

- yaSSL library, 546

Z

- Zabbix tool, 588

- Zenoss tool, 588

- ZFS filesystem, 332, 333

- Zmanda Recovery Manager (ZRM), 515–516

- ZRM (Zmanda Recovery Manager), 515–516

- Zwicky, Elizabeth (*Building Internet Firewalls*), 542

作者简介

Baron Schwartz 是一名软件工程师，他住在弗吉尼亚州的 Charlottesville，在网上用的名字是 Xaprb，这是他名字的第一部分按 QWERTY 键盘的顺序打在 Dvorak 键盘上时显示出来的名字。当他不忙于解决有趣的编程挑战时，Baron 就会和他的妻子 Lynn、狗 Carbon 一起享受闲暇时光。他的关于软件工程的博客地址是 <http://www.xaprb.com/blog>。

Peter Zaitsev，MySQL AB 公司高性能组的前任经理，现正运作着 mysqlperformanceblog.com 网站。他擅长于帮助管理员为每天有着数以百万计访问量的网站修补漏洞，使用数百台服务器来处理 TB 级的数据。他常常为了找到一个解决方案而修改和升级软硬件（比如查询优化）。Peter 还经常在讨论会上发表演讲。

Vadim Tkachenko，Percona 公司的合伙人，该公司是一家专业的 MySQL 性能咨询公司。他过去是 MySQL AB 公司的性能工程师。作为一名在多线程编程和同步领域里的专家，他的主要工作是基准测试、特征分析和找出系统瓶颈。他还在性能监控和调优方面做着一些工作，使 MySQL 在多个 CPU 上更具有伸缩性。

Jeremy D. Zawodny 和他的两只猫在 1999 年底从俄亥俄州的西北部搬到了硅谷，这样他就能为 Yahoo! 工作了——那时他刚好亲眼见证了 .com 泡沫的破灭。他在 Yahoo! 工作了八年半，将 MySQL 和其他开源技术组合起来使用，找到有趣的、令人兴奋的用途，而它们往往也是很大的用途。

近段时间，他重新发掘出了对飞行的热爱。其实，早在 2003 年年初，他就已经取得了私人滑翔机飞行员的执照，2005 年获得商业飞行员的定级。从那时起，他花了大量的空闲时间驾驶滑翔机，飞翔在 Hollister、加利福尼亚和 Tahoe 湖地区上空。他偶尔还会驾驶单引擎轻型飞机，和别人共同拥有一架 Citabria 7KCAB 和一架 Cessna 182。临时的咨询工作可以帮助他支付飞行账单。

Jeremy 和他可人的妻子及四只猫生活在加州的旧金山湾区。他的博客地址是 jeremy.zawodny.com/blog。

Arjen Lentz 出生在阿姆斯特丹，但从千禧年以来他和他美丽的女儿 Phoebe、黑猫 Figaro 一直生活在澳大利亚的 Queensland。Arjen 最初是 C 程序员，在 MySQL AB 公司(2001-2007)里是第 25 号职员。在 2007 年短暂的休息之后，Arjen 创建了 Open Query (<http://openquery.com.au>)，该公司致力于在亚太及临近地区开发和提供数据管理培训和咨询服务。Arjen 也经常在讨论会和用户群中发表讲演。在充裕的闲暇时间里，Arjen 热衷于烹饪、园艺、阅读、露营，以及研究 RepRap。他的博客地址是 <http://arjen-lentz.livejournal.com>。

Derek J. Balling 自 1996 年以来就一直 Linux 系统管理员。他协助 Yahoo! 那样的公司和 Vassar 学院那样的机构建立和维护服务器基础设施，也曾为 Perl 杂志和其他一些在线杂志撰写文章，并一直为 LISA (Large Installation System Administration) 会议的编程委员会服务。目前，他作为数据中心经理受雇于 Answers.com。

当不做与计算机有关的事情时，Derek 喜欢和他的妻子 Debbie 及他们的动物群（四只猫和一只狗）在一起。在博客 <http://blog.megacity.org> 上，他也会对当前热点发出评论或写些近来惹恼他的事情。

封面说明

High Performance MySQL 的封面动物是一只雀鹰(*Accipiter nisus*)，它是猎鹰家族的一员，生活在欧亚大陆和北非的林地周围。雀鹰有一条长长的尾巴和一双短翅膀；雄鸟是蓝灰色的，有一个浅棕色的胸部；雌鸟大多是棕灰色的，胸部几乎全白。雄鸟（28 厘米）通常要比雌鸟（38 厘米）小一些。

雀鹰生活在针叶林里，以小型哺乳动物、昆虫和鸟类为食。它们的巢一般筑在树上，有时甚至在悬崖峭壁上。每年夏初，在最高一棵树的主干上的巢里，雌鸟产下 4 至 6 个白色的，略带红色和棕色斑点的蛋。而雄鸟会给雌鸟和孩子们喂食。

像所有的老鹰一样，雀鹰具有在飞行时突然高速俯冲的能力。无论是高飞还是滑翔，雀鹰都会有带着明显特征的拍翅－拍翅－滑行的动作；它的大尾巴使它能够扭身，轻松地出入树林。

封面图片是一幅 19 世纪雕版画，来自于 Dover Pictorial Archive。